



**Уральский  
федеральный  
университет**

имени первого Президента  
России Б.Н.Ельцина

**Институт радиоэлектроники  
и информационных  
технологий**

**И. А. СЕЛИВАНОВА**

**В. А. БЛИНОВ**

# ПОСТРОЕНИЕ И АНАЛИЗ АЛГОРИТМОВ ОБРАБОТКИ ДАННЫХ

Учебно-методическое пособие

Министерство образования и науки Российской Федерации  
Уральский федеральный университет  
имени первого Президента России Б. Н. Ельцина

**И. А. Селиванова, В. А. Блинов**

# **ПОСТРОЕНИЕ И АНАЛИЗ АЛГОРИТМОВ ОБРАБОТКИ ДАННЫХ**

Учебно-методическое пособие

*Рекомендовано методическим советом УрФУ для студентов,  
обучающихся по программе бакалавриата  
по направлению подготовки  
230100 — Информатика и вычислительная техника*

Екатеринбург  
Издательство Уральского университета  
2015

УДК 004.021(075.8)  
ББК 32.973-018.2я73  
С29

Рецензенты:

кафедра прикладной информатики Института урбанистики УралГАХА  
(зав. кафедрой канд. техн. наук, доц. *Г. Б. Захарова*);

зав. кафедрой математики и естественно-научных дисциплин Уральского института экономики, управления и права, канд. физ.-мат. наук, доц. *С. П. Трофимов*

Научный редактор — канд. техн. наук, доц. *В. П. Битюцкий*

**Селиванова, И. А.**

С29 Построение и анализ алгоритмов обработки данных:  
учеб.-метод. пособие / И. А. Селиванова, В. А. Блинов. —  
Екатеринбург : Изд-во Урал. ун-та, 2015. — 108 с.  
ISBN 978-5-7996-1489-8

Пособие содержит необходимый теоретический материал и примеры реализации данных, которые используются для неформального описания и реализации алгоритмов. Приведены и исследованы алгоритмы внутренней и внешней сортировки, алгоритмы поиска. Все рассмотренные методы сопровождаются наглядными примерами, кроме того, для большинства алгоритмов приведены фрагменты программного кода, что облегчает понимание деталей реализации алгоритмов. Заключительная глава содержит задания для лабораторных занятий.

Библиогр.: 4 назв. Табл. 6. Рис. 46.

УДК 004.021(075.8)  
ББК 32.973-018.2я73

ISBN 978-5-7996-1489-8

© Уральский федеральный  
университет, 2015

## 1. АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Основой программирования является алгоритм — конечный набор инструкций, приводящий от начальных данных к искомому результату. Алгоритмы необходимы из-за того, что решение задач ведется при ограниченном наборе ресурсов, таких как память, вычислительное время и прочее.

На данный момент разработано множество алгоритмов различной сложности и в различных областях, и знание алгоритмов, их использование и комбинирование позволяет решить большинство современных задач программирования.

Структурой данных называют множество элементов данных и множество связей между ними. Как и в случае с алгоритмами, их существует множество, и каждая используется в своей области задач.

Для реализации многих приложений выбор структуры данных — единственное важное решение: когда выбор сделан, разработка алгоритмов не вызывает затруднений. Для одних и тех же данных различные структуры будут занимать неодинаковое дисковое пространство. Одни и те же операции с различными структурами данных создают алгоритмы неодинаковой эффективности. Поэтому выбор алгоритмов тесно взаимосвязан с выбором структуры данных.

### *1.1. Классификация структур данных*

Классификация структур данных может быть выполнена по различным признакам:

- по способу представления;
- по виду памяти, используемой для сохранности данных;
- по сложности представления;

- по характеру упорядоченности элементов в структуре;
- по изменчивости.

Способ представления. Различают физическую и логическую (абстрактную) структуры данных. Физическая структура данных — это способ представления (хранения) данных в машинной памяти компьютера. Логическая структура — это рассмотрение структуры данных без учета ее представления в машинной памяти. В общем случае между логической и соответствующей ей физической структурами существуют расхождения, которые определяются самой структурой и особенностями той среды, в которой она должна быть реализована. Существуют процедуры, которые осуществляют отображение логической структуры в физическую и, наоборот, физической структуры в логическую.

Вид памяти, используемой для сохранности данных. В зависимости от размещения физических структур, а соответственно, и доступа к ним, различают внутренние и внешние структуры данных. Внутренние структуры — это данные, размещенные в статической и динамической памяти компьютера. Внешние структуры размещаются на внешних устройствах и называются файловыми структурами или файлами. Примеры файловых структур: файлы, *B*-деревья и др.

Сложность представления. Структуры данных делятся на элементарные и составные. Элементарными называются такие структуры данных, которые не могут быть расчленены на составные части. С точки зрения физической структуры важным является то обстоятельство, что в конкретной машинной архитектуре, в конкретной системе программирования всегда можно заранее сказать, каков будет размер элементарной единицы данных и как она будет размещена в памяти. С логической точки зрения элементарные единицы данных являются неделимыми. Составными называются такие структуры данных, которые могут быть разбиты на части — другие (элементарные или составные) структуры данных. Составные структуры данных конструируются программистом с использованием средств интеграции данных, предоставляемых языками программирования.

Характер упорядоченности элементов в структуре. Важный признак составной структуры данных — характер упорядоченности ее частей. По этому признаку структуры можно делить на линейные и нелинейные. Линейные структуры разделяют на структуры с последовательным расположением элементов в памяти (векторы, строки,

массивы, стеки, очереди) и структуры с произвольным связным распределением элементов в памяти, к которым относятся односвязные и двусвязные линейные списки. Нелинейные структуры — много-связные списки, деревья, графы.

**Изменчивость.** Изменчивость определяется изменением числа элементов и/или связей между составными частями структуры. В определении изменчивости структуры не учитывается факт изменения значений элементов данных, поскольку в этом случае все структуры данных имели бы свойство изменчивости. По признаку изменчивости различают структуры статические и динамические. Статическими, например, являются массивы, множества, записи, таблицы; динамическими — связные списки, графы, деревья.

### *1.2. Операции над структурами данных*

Набор операций, используемых во многих алгоритмах, ограничивается возможностью вставлять элементы в множество, удалять их, а также проверять, принадлежит ли множеству тот или иной элемент. Динамическое множество, поддерживающее перечисленные операции, называется словарем (dictionary).

В динамических множествах некоторых типов предполагается, что одно из полей объекта идентифицируется как ключевое (key field). Если все ключи различны, то динамическое множество представимо в виде набора ключевых значений. Иногда объект содержит сопутствующие данные (satellite data), которые находятся в других его полях, но не используются реализацией множества. Кроме того, объект может содержать поля, доступные для манипуляции во время выполнения операций над множеством; иногда в этих полях хранятся данные или указатели на другие объекты множества.

На множестве структур данных существует набор операций, разделяющийся на запросы (queries), которые возвращают информацию о множестве, и модифицирующие операции (modifying operations), изменяющие множество. Ниже приведен список типичных операций:

- поиск в множестве элемента с заданным значением;
- добавление в множество элемента с заданным значением;
- удаление из множества элемента с заданным значением;
- поиск в множестве элемента с наименьшим значением;
- поиск в множестве элемента с наибольшим значением.

Время, необходимое для выполнения операций множества, обычно измеряется в единицах, связанных с размером множества, который указывается в качестве одного из аргументов (например,  $O(\log n)$ ).

### *1.3. Понятие алгоритма обработки данных*

Алгоритмом обработки данных называют метод решения задачи, который возможно реализовать в выбранной среде программирования. Тщательная разработка алгоритма является весьма эффективной частью процесса решения задачи в любой области применения. В разработку алгоритма для реальной задачи входит осознание степени ее сложности, выяснение ограничений на входные данные, разбиение задачи на менее трудоемкие подзадачи.

Алгоритм не должен быть привязан к конкретной реализации. В силу разнообразия используемых сред программирования, их требований к аппаратным ресурсам и платформенной зависимости, сходные по структуре, но различные в реализации алгоритмы могут выдавать отличающиеся по эффективности результаты. При этом некоторые среды программирования содержат встроенные библиотечные функции, реализующие базовые алгоритмы обработки данных. Чтобы решения были переносимыми и оставались актуальными, не рекомендуется их ориентировать на процедурную реализацию среды. Поэтому главным в рассматриваемом подходе является выбор метода решения с учетом специфики задачи. Адаптация к среде осуществляется позже.

Выбор того или иного метода обработки данных определяется не только сложностью задачи. Учитывать необходимо и массовость применения разработанного кода: при однократном или редком обращении к реализации предпочтительнее бывают простые алгоритмы, которые несложны в разработке. При этом допускается увеличение времени работы программы.

Алгоритмы должны обладать следующими формальными свойствами:

- понятность;
- дискретность;
- детерминированность;
- результативность;
- массовость.

**Понятность.** Исполнитель алгоритма должен знать, как его выполнять.

**Дискретность.** Алгоритм должен представлять процесс решения задачи как последовательное выполнение простых шагов (этапов), каждый из которых выполняется за конечное время.

**Детерминированность.** Каждое правило алгоритма должно быть четким, однозначным и выдавать для одних исходных данных всегда один и тот же результат.

**Результативность.** При корректно заданных исходных данных алгоритм должен завершать работу и выдавать результат за конечное число шагов.

**Массовость.** Алгоритм разрабатывается в общем виде и должен быть применим для некоторого класса задач и разных исходных данных.

### *1.4. Представление алгоритмов*

На практике наиболее распространены следующие формы представления алгоритмов:

- словесная (записи на естественном языке);
- графическая (изображения из графических символов);
- псевдокоды (полуформализованные описания алгоритмов на условном алгоритмическом языке, включающие в себя как элементы языка программирования, так и фразы естественного языка, общепринятые математические обозначения и др.);
- программная (тексты на языках программирования).

Словесный способ записи алгоритмов представляет собой описание последовательных этапов обработки данных. Алгоритм задается в произвольном изложении на естественном языке. Например: записать алгоритм нахождения наибольшего общего делителя (НОД) двух натуральных чисел.

Алгоритм может быть следующим:

- 1) задать два числа;
- 2) если числа равны, то взять любое из них в качестве ответа и остановиться, в противном случае продолжить выполнение алгоритма;
- 3) определить большее из чисел;



- 4) заменить большее из чисел разностью большего и меньшего из чисел;
- 5) повторить алгоритм с шага 2.

Словесный способ не имеет широкого распространения из-за отсутствия строгой формализации словесного описания алгоритма.


Графический способ представления алгоритмов является более компактным и наглядным по сравнению со словесным. При графическом представлении алгоритм изображается в виде последовательности связанных между собой функциональных блоков, каждый из которых соответствует выполнению одного или нескольких действий. Такое графическое представление называется схемой алгоритма (или граф-схемой алгоритма — ГСА).





В схеме алгоритма каждому типу действий (вводу исходных данных, вычислению значений выражений, проверке условий, управлению повторением действий, окончанию обработки и т. п.) соответствует геометрическая фигура, представленная в виде блочного символа. Блочные символы соединяются линиями переходов, определяющими очередность выполнения действий.

Правила выполнения схем алгоритма определяются в ГОСТ 19.701–90. В следующей таблице приведены наиболее часто употребляемые символы.

Таблица 1.1

Элементы схемы алгоритма

Название символа	Символ	Отображаемая функция
Процесс		Вычислительное действие или последовательность действий. Выполнение одной или нескольких операций, обработка данных любого вида (изменение значения данных, формы представления, расположения). Для улучшения наглядности схемы несколько отдельных блоков обработки можно объединять в один блок. Представление отдельных операций достаточно свободно

Решение		<p>Логический блок: проверка условий.</p> <p>Выбор направления выполнения алгоритма в зависимости от некоторых условий.</p> <p>Используется для обозначения переходов управления по условию. В каждом блоке «решение» должны быть указаны вопрос, условие или сравнение, которые он определяет.</p> <p>Отображает решение или функцию с одним входом и несколькими альтернативными выходами, например знаки <math>&gt;</math>, <math>&lt;</math>, <math>=</math>, операции if, case</p>
Блоки ввода-вывода		Общее обозначение ввода или вывода данных
		Вывод данных, носителем которых служит документ
Терминатор		<p>Начало или конец программы, пуск-останов, вход или выход в подпрограммах.</p> <p>Элемент отображает выход во внешнюю среду и вход из внешней среды (начало и конец программы)</p>
Предопределенный процесс		Используется для указания обращений к вспомогательным алгоритмам, существующим автономно в виде некоторых самостоятельных модулей, и для обращений к библиотечным подпрограммам
Модификация		<p>Начало цикла.</p> <p>Используется для организации циклических конструкций. Внутри блока записывается параметр цикла, для которого указываются его начальное значение, граничное условие и шаг изменения значения параметра для каждого повторения</p>

На рис. 1.1 изображена типовая схема алгоритма.

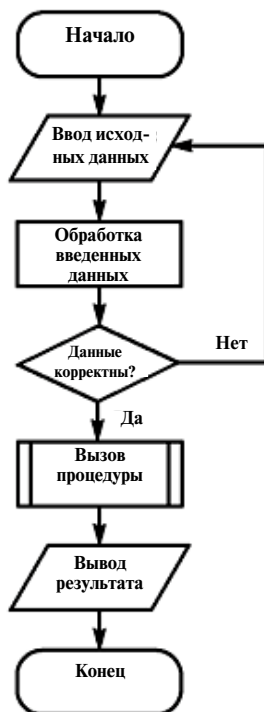


Рис. 1.1. Типовая схема алгоритма

Псевдокод занимает промежуточное место между естественным и формальным языками. С одной стороны, он близок к обычному естественному языку, поэтому алгоритмы могут на нем записываться и читаться как обычный текст. С другой стороны, в псевдокоде используются некоторые формальные конструкции и математическая символика, что приближает запись алгоритма к общепринятой математической записи.

В псевдокоде не приняты строгие синтаксические правила для записи команд, присущие формальным языкам, что облегчает запись алгоритма на стадии его проектирования и дает возможность использовать более широкий набор команд, рассчитанный на абстрактного исполнителя.

Другое различие между псевдокодом и обычным кодом заключается в том, что в псевдокоде, как правило, не рассматриваются не-

которые вопросы, которые приходится решать разработчикам программного обеспечения. Такие вопросы, как абстракция данных, модульность и обработка ошибок часто игнорируются, чтобы более выразительно передать суть алгоритма.

До сих пор не принята какая-либо форма псевдокода в качестве стандарта. Главная цель использования псевдокода — обеспечить понимание алгоритма человеком, сделать описание более воспринимаемым, чем исходный код на языке программирования.

Пример записи алгоритма на псевдокоде. В данном примере на псевдокоде записан алгоритм поиска минимального элемента в массиве. На его вход подается массив  $s[0..n]$ , содержащий последовательность из  $n$  чисел. По окончании работы алгоритма в переменной  $min$  содержится минимальное значение элемента массива.

1. АЛГ
2. НАЧ
3. ЦЕЛ  $n, min$ ;
4. МАССИВ  $s[n]$ ;
5. ЦЕЛ  $i$ ;
6.  $min = s[0]$ ;
7. НЦ ( $i=0; i < n; i++$ )
8.       ЕСЛИ ( $s[i] < min$ )
9.       ТО  $min = s[i]$ ;
10. КЦ
11. ВЫВОД  $min$ ;
12. КОН

Простейший вариант программы на языке C поиска минимального элемента массива приведен ниже.

```
#include <stdio.h>
int main () {
    int min, s [7] = {15,28,3,-5,-17,8,1};
    min = s [0];
    for (int i=0; i<7; i++) {
        if (s [i] <min)
            min = s [i];
    }
    printf («%d\n», min);
    return (0);
}
```

## 2. АНАЛИЗ СЛОЖНОСТИ АЛГОРИТМОВ

### *2.1. Классы сложности алгоритмов*

В процессе решения прикладных задач выбор подходящего алгоритма вызывает определенные трудности. Алгоритм должен удовлетворять следующим противоречащим друг другу требованиям: быть простым для понимания, перевода в программный код и отладки;

эффективно использовать вычислительные ресурсы и выполняться по возможности быстро.

Если разрабатываемая программа, реализующая некоторый алгоритм, должна выполняться всего несколько раз, то первое требование наиболее важно. В этом случае стоимость программы оптимизируется по стоимости написания (а не выполнения) программы. Если решение задачи требует значительных вычислительных затрат, то стоимость выполнения программы может превысить стоимость написания программы, особенно если программа выполняется многократно. Поэтому более предпочтительным может стать сложный комплексный алгоритм (в надежде, что результирующая программа будет выполняться существенно быстрее). Таким образом, прежде чем принимать решение об использовании того или иного алгоритма, необходимо оценить сложность и эффективность этого алгоритма.

Сложность (трудоемкость) алгоритма — это величина, отражающая порядок величины требуемого ресурса (времени или дополнительной памяти) в зависимости от размерности задачи.

Таким образом, будем различать временную  $T(n)$  и пространственную  $V(n)$  сложности алгоритма. Чаще при рассмотрении оценок сложности используется только временная сложность.

Временная сложность алгоритма определяется количеством входных данных. Для простоты входные данные представляются параметром  $n$ . Этот параметр пропорционален величине обрабатываемого набора данных и может обозначать:

- размер массива или файла при сортировке или поиске;
- степень полинома;
- количество символов в строке;
- другую абстрактную меру объема рассматриваемой задачи.

Параметр  $n$  используется для выражения ресурсных требований программ и оценки времени их выполнения с использованием математических формул. Обычно говорят, что временная сложность алгоритма имеет порядок  $T(n)$  от входных данных размера  $n$ . Точно определить величину  $T(n)$  на практике представляется довольно трудно. Поэтому прибегают к асимптотическим отношениям с использованием  $O$ -символики.

Например, если число тактов (действий), необходимое для работы алгоритма, выражается как  $1,5n^2 + 7n \cdot \log n + 3n + 4$ , то это алгоритм, для которого  $T(n)$  имеет порядок  $O(n^2)$ . Фактически, порядок представляет собой показатель старшей степени многочлена. При использовании обозначения  $O(\cdot)$ , имеют в виду не точное время исполнения, а только его предел сверху — верхнюю асимптотическую границу. Если, например, алгоритму требуется время порядка  $O(n^2)$ , имеют в виду, что время исполнения задачи растет не быстрее, чем квадрат количества элементов. Для примера приведем числа, иллюстрирующие скорость роста для нескольких функций, которые часто используются при оценке временной сложности алгоритмов (табл. 2.1).

Таблица 2.1  
Скорость роста часто используемых функций оценки  
временной сложности алгоритмов

$n$	$\log n$	$n \cdot \log n$	$n^2$
1	0	0	1
16	4	64	256
256	8	2048	65 536
4096	12	49 152	16 777 216
65 536	16	1 048 565	4 294 967 296
1 048 476	20	20 969 520	1 099 301 922 576
16 775 616	24	402 614 784	281 421 292 179 456

Если считать, что числа соответствуют микросекундам, то для задачи с 1048476 элементами алгоритму со временем работы  $O(\log n)$  потребуется 20 микросекунд, а алгоритму со временем работы  $O(n^2)$  — более 12 дней.

Таблица 2.2

Классы сложности алгоритмов в зависимости  
от функции трудоемкости

Вид $f(n)$	Характеристика класса алгоритмов
1	Большинство инструкций большинства функций запускается один или несколько раз. Если все инструкции программы обладают таким свойством, то время выполнения программы <i>постоянно</i>
$\log n$	Такое время выполнения обычно присуще программам, которые сводят большую задачу к набору меньших подзадач, уменьшая на каждом шаге размер задачи на некоторый постоянный фактор. Изменение основания не оказывает заметного влияния на изменение значения логарифма: при $n = 1000$ , $\log_{10} n = 3$ , $\log_2 n \approx 10$
$n$	Когда время выполнения программы является <i>линейным</i> , это обычно значит, что каждый входной элемент подвергается небольшой обработке
$n \log n$	Время выполнения, пропорциональное $n \log n$ , возникает тогда, когда алгоритм решает задачу, разбивая ее на меньшие подзадачи, решая их независимо и затем объединяя решения
$n^2$	Когда время выполнения алгоритма является <i>квадратичным</i> , он полезен для практического использования при решении относительно небольших задач. Квадратичное время выполнения обычно появляется в алгоритмах, которые обрабатывают все пары элементов данных (возможно, в цикле двойного уровня вложенности)
$n^3$	Похожий алгоритм, который обрабатывает тройки элементов данных (наиболее часто в цикле тройного уровня вложенности), имеет <i>кубическое</i> время выполнения и практически применим лишь для малых задач
$2^n$	Лишь несколько алгоритмов с <i>экспоненциальным</i> временем выполнения имеют практическое применение, хотя такие алгоритмы возникают естественным образом при попытках прямого решения задачи, например полного перебора

Алгоритмы делятся на классы не только по времени выполнения, но и в зависимости от дополнительно подключаемой памяти. Кроме того, существует разделение на классы задач, которые решаются детерминированными и недетерминированными алгоритмами. Вторые отличаются тем, что решают задачи некоторого выбора из предложенных вариантов. В такой классификации существуют следующие классы:

- *класс  $P$*  — класс детерминированных полиномиальных алгоритмов, функция трудоемкости которых определяется от входных параметров как  $O(n)$ ,  $O(n^2)$ ,  $O(n^3)$  и т. д. Примеры задач, решаемых за полиномиальное время: умножение матриц, сортировка массива;
- *класс  $L$*  — класс детерминированных алгоритмов, использующих дополнительно  $O(\log n)$  памяти;
- *класс  $NL$*  — класс недетерминированных алгоритмов, использующих дополнительно  $O(\log n)$  памяти. Класс  $L \in NL$ . Пример задачи  $NL$  — нахождение пути в графе. Задачи класса  $NL$  обычно решаются за полиномиальное время;
- *класс  $NP$*  — класс недетерминированных полиномиальных алгоритмов. Задача о равенстве классов  $P$  и  $NP$  является одной из самых актуальных задач теории алгоритмов. Примером задач  $NP$  являются задача о мешке и задача коммивояжера.

## 2.2. Методы оценки сложности алгоритмов

Основными алгоритмическими конструкциями в процедурном программировании являются следование, ветвление и цикл. Для получения функций трудоемкости для лучшего, среднего и худшего случаев при фиксированной размерности входа необходимо учесть различия в оценке основных алгоритмических конструкций:

- Трудоемкость следования есть сумма трудоемкостей блоков, следующих друг за другом:

$$F = F_1 + F_2 + F_3 + \dots + F_n.$$

- Трудоемкость ветвления определяется вероятностью перехода к каждой из инструкций исходя из условия, проверка условия также имеет трудоемкость:

$$F_{if} = F_1 + F_{then} \cdot P + F_{else} \cdot (1 - P),$$

где  $P$  и  $(1 - P)$  — это вероятности перехода по условиям *then* и *else* соответственно.



- Трудоемкость цикла зависит от его вида, для цикла от 1 до  $n$  с параметрами будет справедливой формула:

$$F_{\text{for}} = 1 + 3 \cdot n + n \cdot F.$$

Слагаемое  $1 + 3 \cdot n$  определяет трудоемкость входа в цикл, проверки условия и инкрементации, слагаемое  $n \cdot F$  определяет трудоемкость непосредственно тела цикла.

- Реализация цикла с предусловием и с постусловием не меняет методики оценки его трудоемкости. На каждом проходе выполняется оценка трудоемкости условия, изменения параметров (при наличии) и тела цикла. Общие рекомендации для оценки циклов с условиями затруднительны, так как в значительной степени зависят от исходных данных.
- В случае использования вложенных циклов их трудоемкости перемножаются.

Таким образом, для оценки трудоемкости алгоритма может быть сформулирован общий метод получения функции трудоемкости.

Декомпозиция алгоритма предполагает выделение в алгоритме базовых конструкций и оценку их трудоемкости. При этом рассматривается следование основных алгоритмических конструкций.

Построчный анализ трудоемкости по базовым операциям языка подразумевает либо совокупный анализ (учет всех операций), либо пооперационный анализ (учет трудоемкости каждой операции).

Обратная композиция функции трудоемкости на основе методики анализа базовых алгоритмических конструкций для лучшего, среднего и худшего случаев.

Особенностью оценки ресурсной эффективности рекурсивных алгоритмов является необходимость учета дополнительных затрат памяти и механизма организации рекурсии. Поэтому трудоемкость рекурсивных реализаций алгоритмов связана с количеством операций, выполняемых при одном рекурсивном вызове, а также с количеством таких вызовов. Учитываются также затраты на возвращение значений и передачу управления в точку вызова.

$$F = 2 \cdot (P + K + R + L + 1),$$

где  $P$  — количество передаваемых фактических параметров,  $R$  — количество сохраняемых в стеке регистров,  $K$  — количество возвращаемых по адресной ссылке значений,  $L$  — количество локальных ячеек функции, а дополнительная единица учитывает операции с адресом возврата.

Оценка требуемой памяти стека может быть получена следующим образом: так как рекурсивные вызовы обрабатываются последовательно, то в конкретный момент времени в стеке хранится не фрагмент дерева рекурсии, а цепочка рекурсивных вызовов — унарный фрагмент дерева. Поэтому объем стека определяется максимально возможным числом одновременно полученных рекурсивных вызовов.

Анализ совокупной трудоемкости рекурсивного алгоритма можно выполнять разными способами в зависимости от формирования итоговой суммы базовых операций: по цепочкам рекурсивных вызовов и возвратов, по вершинам рекурсивного дерева.

Пример 1. Оценка временной сложности функции пузырьковой сортировки.

```
void BubbleSort (int n, int* x) {
    int buf = 0;
    for (int i = n - 1; i >= 0; --i)
        for (int j = i - 1; j >= 0; --j)
            if (x[j] > x[j + 1]) {
                buf = x[j];
                x[j] = x[j + 1];
                x[j + 1] = buf;
            }
}
```

В худшем случае исходные данные должны быть отсортированы в обратном порядке. Тогда обмен будет выполняться каждый раз, от  $(n - 1)$  раза на первом шаге до 1 раза на последнем. Тогда по формуле суммы  $n$  членов арифметической прогрессии рассчитаем количество пересылок:  $\frac{(n-1)+1}{2}(n-1) = \frac{n(n-1)}{2}$ . В общем случае условие

может выполняться на очередном шаге от 1 до  $(n - 1)$  раз, то есть в среднем на каждом шаге произойдет  $n/2$  обменов. В лучшем случае массив уже отсортирован, и не произойдет ни одного обмена, но произойдет  $\frac{n(n-1)}{2}$  проверок условия. Соответственно, сложность алгоритма  $O(n^2)$ .

Пример 2. Оценка временной сложности формулы расчета биномиального коэффициента.

$$C_n^m = \frac{n!}{m!(n-m)!} (n \geq m)$$

Ниже приведена программная реализация функции расчета биномиального коэффициента на основе рекурсии.

```
int Binom (int n, int m) {  
    if (m == 0) return 1; //База рекурсии  
    return Binom (n - 1, m - 1) * n / m; //Декомпозиция  
}
```

В худшем случае  $m = n$ . Тогда будет выполнено  $(n + 1)$  обращений к функции, которая выполнит в  $n$  случаях три операции, а в одном — возвратит значение. Функция при каждом обращении передает два параметра, не использует локальных переменных, а при возвращении  $(n + 1)$  раз передает управление в точку вызова. Соответственно, сложность алгоритма в худшем случае составит  $O(n)$  или  $O(m)$ .

В среднем случае  $0 < m < n$ . При этом количество рекурсивных вызовов составит  $(m + 1)$ . Соответственно сложность алгоритма в среднем случае составит  $O(m)$ .

Лучший случай достигается при  $m = 0$ , когда выполняется единственный вызов функции, передача двух параметров и возвращение в точку вызова, то есть оценка лучшего случая  $O(1)$ .

### 3. СТРУКТУРЫ ДАННЫХ

---

#### 3.1. Типы данных

Все обрабатываемые компьютером данные в конечном счете разбиваются на отдельные биты. Однако написание программ, обрабатывающих исключительно биты, — слишком трудоемкое занятие. Поэтому в программировании введено такое понятие, как *тип данных*. Тип данных определяет множество значений, которые могут иметь константы, переменные, выражения, результаты операций и возвращаемые значения функций.

Программы обрабатывают информацию, которая происходит из математических или естественных языковых описаний окружающего мира. Вычислительная среда обеспечивает встроенную поддержку основных строительных блоков подобных описаний — чисел и символов. Таким образом, в качестве базовых типов данных практически во всех языках программирования используются:

- целые числа (integer);
- числа с плавающей точкой (float);
- символы (character).

Диапазон значений целочисленных типов данных зависит от количества бит, используемого для их представления. Числа с плавающей точкой представляют собой приближение к действительным числам, а используемое для их представления количество бит определяет точность этого приближения. Стоит также отметить, что числовые типы данных бывают со знаком и без знака. Например, если для целочисленного типа данных отведено 4 байта, то он может иметь диапазон  $[-2^{31}, 2^{31} - 1]$  или диапазон  $[0, 2^{32} - 1]$ .

Кроме базовых типов данных, описанных в языках программирования, существуют также и пользовательские типы данных, которые

создают программисты и которые обычно используются для решения более узкого круга задач.

Тип данных определяет не только множество значений, но и набор операций над этими значениями. Именно операции связаны с типами, а не наоборот. При выполнении операции необходимо обеспечить, чтобы ее операнды и результат отвечали фиксированному типу. Если это не выполняется, то необходимо выполнить преобразование типа. Бывает приведение (англ. casting), или явное преобразование типов, и неявное преобразование типов, выполняемое компилятором в соответствии с правилами языка программирования. Например, в языке C выражение  $((\text{float}) x) / N$ , где  $x$  и  $N$  — целые числа, включает в себя оба типа преобразований: оператор  $(\text{float})$  выполняет приведение — величина  $x$  преобразуется в значение с плавающей точкой. Затем для  $N$  выполняется неявное преобразование, так как в соответствии с правилами языка C оба аргумента оператора деления должны представлять значения с плавающей точкой.

В случае, когда требуется обработать не одно значение, а набор данных, используются механизмы группировки (или инкапсуляции) данных. Такими механизмами являются массивы, которые рассматриваются в разделе 3.2, и структуры.

Структурой (англ. structure) называется набор разнородных элементов, расположенных в памяти друг за другом. Каждое поле в структуре определяется типом и уникальным именем. Над структурами можно определять операции, структурами можно описывать пользовательские типы данных. Примером использования структуры является тип данных «точка»:

```
typedef struct {  
    float x;  
    float y;  
} point;
```

Эта структура описывает тип данных, который можно описать как «точка в пространстве с действительными координатами». Обращение к отдельным полям осуществляется по имени:

```
point pt;  
pt.x = 1.0;  
pt.y = 2.0;
```

Над таким типом данных можно, например, определить функцию расчета расстояния между двумя точками:

```
float distance (point, point);
```

Использование структур для хранения большого количества данных одного типа нецелесообразно, поскольку для каждого элемента нужно будет создать собственное поле и обращаться по имени. Для хранения и обработки таких данных используют массивы.

### 3.2. Массив

Массивом называется набор однотипных элементов, расположенных в памяти друг за другом и адресуемых с помощью одного имени и разных индексов.

Массивы используются для обработки большого количества однотипных данных. Имя массива является указателем. Отдельная ячейка данных массива называется элементом массива. Элементами массива могут быть данные любого типа. Массивы могут иметь как одно, так и более одного измерений. В языке С имя массива является указателем на ячейку в памяти, в которой хранится первый элемент массива. Такой подход соответствует организации памяти в компьютере, где слова из памяти извлекаются по адресам. То есть память можно рассматривать как массив, где адреса памяти соответствуют индексам. В зависимости от количества измерений массивы делятся на одномерные, двумерные, трехмерные и так далее до  $n$ -мерного массива. Чаще всего в программировании используются одномерные и двумерные массивы. Массив может быть не только фиксированной, но и произвольной длины (динамический массив). В языке С для реализации динамического массива используются функции выделения памяти `malloc` и `realloc`, применяемые к указателям.

Одномерный массив — массив с одним параметром, характеризующим количество элементов одномерного массива. На рис. 3.1. показана структура целочисленного одномерного массива  $a$ .

8	-9	3	0	5
$a[0]$	$a[1]$	$a[2]$	$a[3]$	$a[4]$

Рис. 3.1. Представление массива

В языке C нумерация массива начинается с 0, так что в данном массиве 5 элементов. Объявляется этот массив следующим образом:

```
int a [5];
```

Здесь `int` — тип данных всех элементов массива, *a* — имя массива, `a [5]` — количество элементов в массиве. При этом адрес переменной *a* в памяти будет указывать на начало массива, а ее фактическим типом является `int *`.

Массивы могут быть инициализированы при объявлении:

```
int a [5] = {8, -9, 3, 0, 5};
```

или

```
int a [5] = {8, -9, 3, 0, 5};
```

Если явно не задать размер массива, то он будет равен количеству инициализированных элементов.

Количество используемых индексов определяет размерность массива:

```
int v [s1];    // объявление одномерного массива
```

```
int A [s1] [s2];    // объявление двумерного массива
```

```
float cube [s1] [s2] [s3];    // объявление трехмерного массива
```

Можно выполнять операции над отдельными элементами массива. Перечень таких операций определяется типом элементов. Доступ к отдельным элементам массива осуществляется через имя массива и индекс (индексы) элемента:

```
cube [0] [0] [10] = 25;
```

```
matrix [10] [30] = cube [0] [0] [10] + 1;
```

В памяти ЭВМ элементы массива обычно располагаются непрерывно, в соседних ячейках. Размер памяти, занимаемой массивом, есть суммарный размер элементов массива.

Примером алгоритма, работающего с массивом, является алгоритм «Решето Эратосфена».

### Алгоритм «Решето Эратосфена»

Алгоритм позволяет найти все простые числа до  $n$ , отсеивая на каждом шаге составные числа из массива  $[1..n]$ .

Суть алгоритма заключается в том, что для каждого следующего простого числа из массива убираются все числа, кратные ему. Таким образом, необходимо выполнить следующие шаги:

1. Выписать подряд все целые числа от двух до  $n$  (2, 3, 4, ...,  $n$ ).
2. Первое простое число  $p = 2$ .
3. Зачеркнуть в списке числа от  $2p$  до  $n$ , считая шагами по  $p$  (это будут числа, кратные  $p$ :  $2p, 3p, 4p, \dots$ ).
4. Найти следующее незачеркнутое число в списке, большее, чем  $p$ , и присвоить значению переменной  $p$  это число.
5. Повторять шаги 3 и 4, пока есть незачеркнутые числа.

Асимптотика алгоритма равна  $O(n \log \log n)$ <sup>1)</sup>.

Первоначальный алгоритм можно оптимизировать. Отметим, что все составные числа до квадрата очередного простого уже должны быть отмечены на предыдущих шагах, поэтому их можно пропустить. Также для нахождения всех простых чисел до  $n$  достаточно просеять массив простыми числами только до  $\sqrt{n}$ . Это уменьшит количество операций примерно в 4 раза. С учетом этих двух оптимизаций приведем текст функции «Решето Эратосфена» на языке C++.

```
bool* EratostheneSieve (int n) {
    bool* prime = (bool*) malloc ((n + 1) * sizeof (bool));
    prime [0] = prime [1] = false;
    for (int i = 2; i * i <= n; ++i) {
        if (prime [i])
            if (i * i <= n)
                for (int j = i * i; j <= n; j += i)
                    prime [j] = false;
    }
    return prime;
}
```

Наконец, можно сразу отбросить все четные числа, кроме 2. Это уменьшит количество операций и объем потребляемой памяти примерно в 2 раза. Однако для такой оптимизации необходимо исполь-

---

<sup>1)</sup> Алгоритмы: построение и анализ / Т. Кормен [и др.]. М. : «Вильямс», 2005.



зовать структуру данных, отличную от массива, поскольку доступ по индексу за  $O(1)$  осуществить не удастся.

Заметим также, что описанные оптимизационные приемы не влияют на асимптотику.

### 3.3. Строка

Строка — это последовательность символов. В языке C строкой является массив символов типа `char`. C-строки также называются нуль-терминированными, поскольку конец информационной части в них обозначается символом ASCII `0x00` (`'\0'`). Так как длина массива задается при его объявлении, то за этим символом вполне может оставаться мусор от предыдущих использований строки. Рассмотрим пример объявления строки:

```
char str [20];
```

Благодаря индексам строки очень похожи на одномерные массивы символов, и доступ к отдельным элементам строки можно получать с использованием этих индексов, выполняя операции, определенные для символьного типа данных.

Однако есть ряд отличий. Операций сравнения строк больше, чем аналогичных операций для массивов: `<`, `>`, `==`, `!=`. Существует операция сцепления (конкатенации) строк `«+»`.

Операции сравнения указывают, какая из двух строк должна первой значиться в словаре. Строки сравниваются посимвольно от начала и до конца. Такой порядок называется лексикографическим. Символы в лексикографическом порядке располагаются в порядке возрастания кода ASCII. При таком сравнении первая же пара различающихся символов определяет, как строка больше. Например, `d f g h < d f g w < f g < z`. Если все символы в строках попарно равны, то говорят, что строки тоже равны. Так, `d f g h = d f g h`.

### 3.4. Связный список

Связный список — структура данных, в каждом элементе (узле) которой хранится информация и один или два указателя на следующий и/или предыдущий элементы структуры.

Подобный подход обеспечивает структурную гибкость — элементы идут в явно заданном порядке и необязательно хранятся в памяти последовательно. В отличие от массивов, получить значение отдельного элемента в списке сразу нельзя, для этого необходимо перебрать все элементы с начала. То есть асимптотика нахождения, добавления и изменения элементов в списке равна  $O(n)$ .

Если элементы списка хранят только одну ссылку, то список называется однонаправленным. Если две ссылки — двунаправленным. Первым элементом в списке часто является фиктивный узел, то есть узел, не содержащий информационной нагрузки. Это позволяет реализовывать пустые списки.

Однонаправленный (односвязный) список — это структура данных, представляющая собой последовательность элементов, в каждом из которых хранится значение и указатель на следующий элемент списка (рис. 3.2). В последнем элементе указатель на следующий элемент равен null. Такой список может быть прямым или обратным. Разница в том, что в обратном списке элементы добавляются перед первым элементом, а не за последним.

**Указатель на первый  
элемент списка**

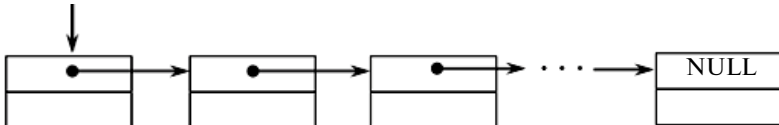


Рис. 3.2. Прямой однонаправленный список

Описание простейшего элемента такого списка:

```
struct node {
    int elem; //Хранимая информация
    node* next; //Указатель на следующий элемент
};
```

Хранимая информация может быть любой, и необязательно должно быть одно информационное поле, их может быть несколько (или его вообще может не быть). Указатель же — обязательная часть элемента списка, и он должен указывать на объект того же типа, что и определяемая структура.

Следующий код создает список из двух элементов, используя структуру `node`, и выводит значение следующего элемента списка из предыдущего:

```
node s = {-9, NULL}, ss; //Объявление элементов, причем элемент
S сразу инициализируется
ss.elem = 40; //Информация в ss
s.next = &ss; //Соединение списка
printf ("%d», s.next->elem); //Вывод значения из ss через указатель в s.
```

Двунаправленный (двусвязный) список — это структура данных, состоящая из последовательности элементов, каждый из которых содержит информационную часть и два указателя на соседние элементы (рис. 3.3). При этом два соседних элемента должны содержать взаимные ссылки друг на друга. В таком списке каждый элемент (кроме первого и последнего) связан с предыдущим и следующим за ним элементами.

Каждый элемент двунаправленного списка имеет два поля с указателями: одно поле содержит ссылку на следующий элемент, другое поле — ссылку на предыдущий элемент, третье поле — информационное. Наличие ссылок на следующее звено и на предыдущее позволяет двигаться по списку от каждого звена в любом направлении: от звена к концу списка или от звена к началу списка, поэтому такой список называют двунаправленным.

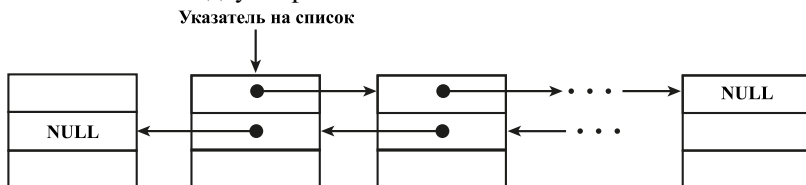


Рис. 3.3. Двунаправленный список

Описание элемента двунаправленного списка:

```
struct node2 {
    int elem;
    node2* next;
    node2* prev;
};
```

В этом случае обязательными являются уже две ссылки — на предыдущий и на следующий элементы.

Примеры команд над связным списком

Добавить элемент сразу после указанного, может использоваться в прямом односвязном или двусвязном списках (пример для односвязного):

```
node *AddAfter (node *s, int a) {
    node *ss = (node*) malloc (sizeof (node));
    ss->elem = a;
    ss->next = s->next;
    s->next = ss;
    return ss;
}
```

Добавить элемент непосредственно перед указанным, может использоваться в обратном односвязном или двусвязном списках (пример для двусвязного):

```
node2 *AddBefore (node2* s, int a) {
    node2 *ss = (node2*) malloc (sizeof (node2));
    ss->next = s;
    ss->elem = a;
    if (s->prev!= NULL) {
        s->prev->next = ss;
        ss->prev = s->prev;
    }
    else ss->prev = NULL;
    s->prev = ss;
    return ss;
}
```

Принцип добавления элемента в связный список изображен на рис. 3.4.

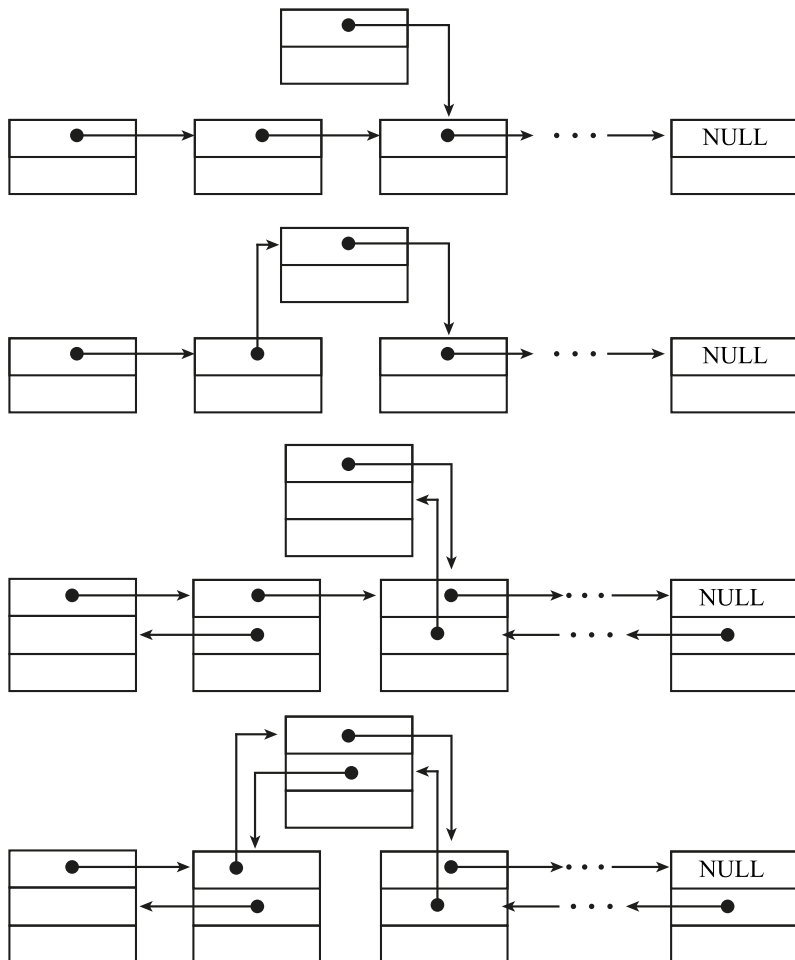


Рис. 3.4. Добавление элемента в односвязном и двусвязном списках

Также обычно требуются такие команды, как: добавить первый или последний элемент, найти элемент или удалить, отсортировать список. Принцип удаления изображен на рис. 3.5.

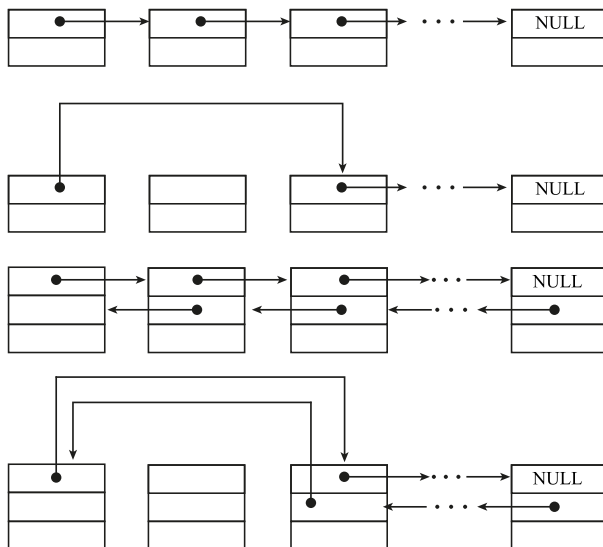


Рис. 3.5. Удаление элемента в односвязном и двусвязном списках

В качестве сортировки в связных списках удобно использовать сортировку вставками (раздел 4.2).

### 3.5. Стек

Стеком (англ. stack — стопка) называется структура данных, в которой можно работать только с одним элементом: тем, который был добавлен последним (принцип Last In First Out — LIFO). Эта структура еще называется стеком магазинного типа. Две основные операции над стеком — добавление (англ. push) и извлечение (англ. pop) верхнего элемента.

Стек, количество элементов в котором ограничено  $n$ , можно организовать на массиве  $A[1..n]$ . У такого массива должен быть атрибут  $\text{top}[A]$ , который хранит индекс последнего добавленного элемента. Если  $\text{top}[A] = 0$ , то стек пустой. Попытки извлечения элемента из пустого стека или добавления элемента в полностью заполненный стек приводят, как правило, к фатальным ошибкам в программах.

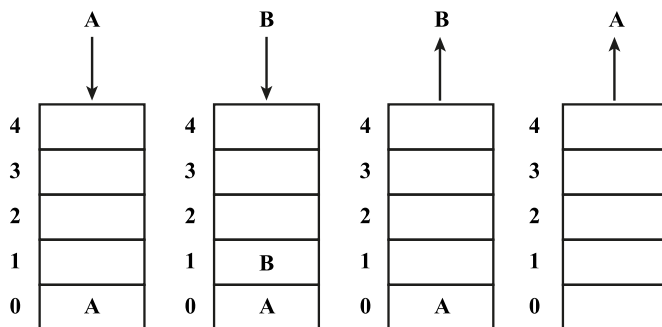


Рис. 3.6. Стек

Стек обычно поддерживает следующие операции:

- *push* — добавить (положить) в конец стека новый элемент;
- *pop* — извлечь из стека последний элемент;
- *peek* — узнать значение последнего элемента, не удаляя его;
- *size* — узнать размер стека;
- *clear* — очистить стек.

Асимптотика этих операций на массиве, очевидно, равна  $O(1)$ .

Пример реализации такого стека:

```
const int max_size = 100;
typedef struct {
    int size;
    int A[max_size];
} stack;

void Push(stack *st, int a) {
    if (st->size < max_size - 1) st->A[++st->size] = a;
};

void Pop(stack *st, int *a) {
    if (st->size >= 0) (*a) = st->A[st->size--];
};
```

Объявленная здесь структура *stack* реализует стек целых чисел. Поле *size* хранит количество элементов в стеке в настоящее время, сами элементы хранятся в элементах массива *A* с индексами  $0..size-1$ . Глобаль-

ная константа `max_size` задает размер создаваемого стека. Функция `Push` помещает элемент  $a$  в стек, а `Pop` удаляет последний элемент из стека и возвращает его значение по указателю.

Перед использованием структуры `stack` необходимо инициализировать:

```
stack st;  
st.size = -1;
```

Следующий код добавляет в стек элемент 5 и затем удаляет его из стека, присваивая значение переменной  $a$ :

```
int a;  
Push (&st, 5);  
Pop (&st, &a);
```

Стек можно реализовать и с помощью списка, обратного односвязного или двусвязного. Такой подход удобен в случае, если в стек помещаются не просто значения или символы, а целые наборы данных. Обратный список желательно использовать для того, чтобы операции добавления и удаления сохраняли асимптотику  $O(1)$ .

Принцип стека используется, например, при расчетах выражений, записанных в обратной польской записи.

#### Польская запись

Польской нотацией (или префиксной нотацией) называется форма записи выражений, при которой оператор располагается слева от операндов. Такая запись называется также бесскобочной, поскольку при детерминированном количестве операндов у каждого оператора скобки не требуются.

Пример.  $(13+10)/2$  в инфиксной форме записи можно представить как  $/(+13\ 10)2$  в префиксной или просто как  $/+13\ 10\ 2$ . Здесь вычисление деления задержится до вычисления результата сложения. Если переставить скобки так, что  $13+(10/2)$ , то префиксное выражение примет вид  $+13/10\ 2$ . Теперь уже сложение задержится до вычисления умножения.

Прямая польская нотация представляет чисто академический интерес и используется в программировании редко. Однако в середине 1950-х годов несколькими учеными был предложен другой вариант



записи, названный обратной польской нотацией (англ. reverse polish notation). Как ясно из названия, эта запись является постфиксной, то есть операнды в ней располагаются перед оператором.

Обратная польская запись имеет ряд преимуществ при вычислении результатов алгебраических выражений: в записи отсутствуют скобки и приоритеты, и принцип вычисления подобен стеку.

Пример. Инфиксное выражение  $(3+4)*6+5$  может быть представлено в постфиксной записи так:  $3\ 4+6*5+$ . В табл. 3.1 приведен пример расчета с использованием стека (указано состояние после выполнения операции, вершина стека выделена).

Таблица 3.1

Пример расчетов в обратной польской нотации

Ввод	Операция	Стек
1	поместить в стек	<b>3</b>
2	поместить в стек	3, <b>4</b>
+	сложение	<b>7</b>
4	поместить в стек	7, <b>6</b>
*	умножение	<b>42</b>
3	поместить в стек	42, <b>5</b>
+	сложение	<b>47</b>

Существует несколько алгоритмов для превращения инфиксных формул в обратную польскую запись. В качестве примера рассмотрим простейший вариант алгоритма, предложенного Э. Дейкстрой и работающего за  $O(n)$  операций. Более полное описание алгоритма можно найти, например, в работе Р. Седжвика<sup>2)</sup>.

Предварительные замечания:

- Исходное арифметическое выражение, содержащее числа, знаки арифметических операций и круглые скобки, рассматривается как входная строка символов.
- Входная строка просматривается слева направо.
- Операнды переписываются в выходную строку, а знаки операций помещаются вначале в стек операций (в стеке могут находиться только знаки операций или открывающая скобка).
- Арифметические операции имеют различный приоритет.

В первую очередь выполняются более приоритетные опера-

<sup>2)</sup> Седжвик Р. Фундаментальные алгоритмы на С. В 5 ч. Ч. 1–4. Анализ ; Структуры данных ; Сортировка ; Поиск. М.: ДиаСофт, 2003.

ции. Операции одного приоритета выполняются подряд, слева направо. Наивысший приоритет имеют операции  $*$ ,  $/$ ,  $\%$ , наименьший приоритет — операции  $+$  и  $-$ .

Алгоритм состоит из следующих шагов:

- Пока есть символы для чтения:
  - Читаем очередной символ.
  - Проводим анализ символа.
    - Если символ является числом, добавляем его к выходной строке.
    - Если символ является открывающей скобкой, помещаем его в стек.
    - Если символ является знаком операции, проверяем, лежит ли сверху в стеке знак операции с большим приоритетом. Если лежит, то «выталкиваем» его из стека в выходную строку. Повторяем, пока в стеке сверху не окажется знак операции с меньшим приоритетом или открывающая скобка и пока стек не пустой. После этого помещаем входной символ в стек.
    - Если символ является закрывающей скобкой, то до тех пор, пока верхним элементом стека не станет открывающая скобка, «выталкиваем» элементы из стека в выходную строку. При этом открывающая скобка удаляется из стека, но в выходную строку не добавляется. Открывающая и закрывающая скобки как бы взаимно уничтожаются и в выходную строку не переносятся. Если стек закончился раньше, чем встретилась открывающая скобка, это означает, что в выражении не согласованы скобки.
  - Если вся входная строка разобрана, «выталкиваем» все символы из стека в выходную строку (в стеке должны были остаться только символы операций; если это не так, значит, в выражении не согласованы скобки).

Расчет выражения в обратной польской нотации описывается алгоритмом:

- Пока есть символы для чтения:
  - Читаем очередной символ.
    - Если символ является числом, помещаем его в стек.
    - Если символ является символом операции, извлекаем из стека два последних числа, производим над ними соответствующую операцию и результат помещаем в стек.

► Когда входная строка закончилась, в стеке должен остаться только один элемент — результат всего выражения.

Ниже приведена программная реализация функций этих алгоритмов на языке C++:

```
bool isop (char in) {
    return (in == «+» || in == «-» || in == «*» || in == «%» || in == «/»);
} //проверяет, является ли символ операцией

int priority (char in) {
    return
        in == «+» || in == «-»? 1:
        in == «*» || in == «/» || in == «%»? 2:
        -1;
} //определяет приоритет операции

string ReversePolishNotation (string Text) {
    stack <char> Sym;
    string revText;
    for (size_t i = 0; i < Text.size (); ++i) {
        if (isalnum (Text [i])) {
            while (i < Text.size () && isalnum (Text [i])) {
                revText += Text [i];
                ++i;
            }
            --i;
            revText += « «;
        }
        else if (isop (Text [i])) {
            if (Sym.size () && isop (Sym.top ()))
                while (priority (Sym.top ()) >= priority
(Text [i])) {
                    revText += Sym.top ();
                    revText += « «;
                    Sym.pop ();
                }
            Sym.push (Text [i]);
        }
    }
}
```

```
        else if (Text [i] == «('» {
            Sym.push (Text [i]);
        }
        else if (Text [i] == «(»') {
            while (Sym.top () != «('» {
                revText += Sym.top ();
                revText += « «;
                Sym.pop ();
            }
            Sym.pop ();
            if (! Sym.empty () && isop (Sym.top ())) {
                revText += Sym.top ();
                revText += « «;
                Sym.pop ();
            }
        }
    }
    while (! Sym.empty ()) {
        revText += Sym.top ();
        if (Sym.size () != 1) revText += « «;
        Sym.pop ();
    }
    return revText;
}
```

**Функция расчета значения выражения:**

```
int RPNCalc (string Text) {
    stack <int> St;
    for (size_t i = 0; i < Text.size (); ++i) {
        if (isalnum (Text [i])) {
            string a;
            while (i < Text.size () && isalnum (Text [i])) {
                a += Text [i];
                ++i;
            }
            --i;
            St.push (atoi (a.c_str ()));
        }
    }
}
```

```
else if (isop (Text [i])) {  
    int r = St.top (); St.pop ();  
    int l = St.top (); St.pop ();  
    switch (Text [i]) {  
        case «+»: St.push (l + r); break;  
        case «-»: St.push (l - r); break;  
        case «*»: St.push (l * r); break;  
        case «/»: St.push (l / r); break;  
        case «%»: St.push (l % r); break;  
    }  
}  
}  
return St.top ();  
}
```

Такой подход позволяет получить само постфиксное выражение и называется «явным». Можно воспользоваться нотацией и в «неявном» виде, располагая операции с помощью стека в таком порядке, что к моменту вычисления очередной операции оба ее операнда уже будут вычислены. Недостатком записи в таком виде является то, что все операции предполагаются бинарными и лево-ассоциативными. Для унарных операций, например, операции отрицания, следует заводить символ, отличный от «-», а для подряд идущих унарных операций или операций возведения в степень учитывать, что они должны обрабатываться справа налево<sup>3)</sup>.

### 3.6. Очередь

Очередью называется структура данных, организованная по принципу First In First Out — FIFO, то есть добавляются элементы только в конец структуры, а извлекаются из начала.

Очередь должна поддерживать те же операции, что и стек. Очередь может быть организована как с помощью массива и двух указателей на начало и конец очереди, так и с помощью двух стеков.

При реализации очереди с использованием массива используются два указателя — head и tail. Head указывает на первый элемент

---

<sup>3)</sup> Головешкин В.А., Ульянов Н.В. Теория рекурсии для программистов. М. : ФИЗМАТЛИТ, 2006.

в очереди, а *tail* — на последний (или следующий за последним). При добавлении элемента *tail* увеличивается на 1, а при удалении элемента *head* увеличивается на 1. Когда каждый из параметров достигает конца массива, его значение сбрасывается. Принцип изображен на рис. 3.7.

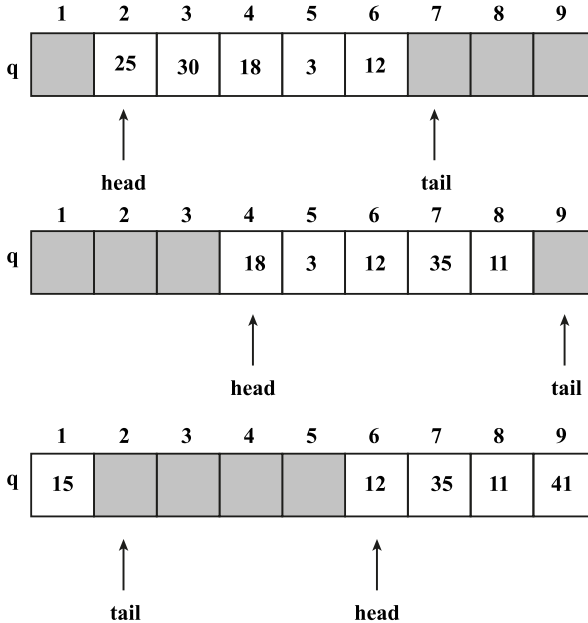


Рис. 3.7. Принцип работы очереди

Ниже приведен фрагмент программы реализации очереди на основе массива.

```
const max_size = 100;
typedef struct {
    int head;
    int tail;
    int A [max_size];
} queue;

void Enqueue (queue *qe, int a) {
    if (qe->tail + 1 < max_size) qe->A [qe->tail++] = a;
```

```
    else {qe->tail = 0; qe->A [max_size — 1] = a;};  
};  
void Dequeue (queue *qe, int *a) {  
    if (qe->head!= qe->tail) {  
        if (qe->head + 1 < max_size) (*a) = qe->A [qe->head++];  
        else {  
            qe->head = 0; (*a) = qe->A [max_size — 1];};  
        }  
};
```

Функции Enqueue и Dequeue — аналоги функций Push и Pop с той разницей, что добавляются элементы в конец, а извлекаются из начала. Переменная head указывает на первый элемент очереди, а tail — на следующую за последним ячейку, в которую, соответственно, будет записываться новый элемент. Таким образом,  $tail > head$  для непустой очереди.

В функциях нет проверок опустошения и переполнения. Очередь на массиве будет переполнена в двух случаях: при  $head = 0$  и  $tail = max\_size - 1$  и при  $head = tail + 1$ .

Реализация очереди на двух стеках:

Процедура enqueue (x):

    S1.push (x)

Процедура dequeue ():

    если S2 пуст:

        если S1 пуст:

            сообщить об ошибке: очередь пуста

    пока S1 не пуст:

        S2.push (S1.pop ())

return S2.pop ()

Разновидностью очереди является так называемая очередь с приоритетом, или priority queue. Над ней определены три основные операции:

- EnqueueWithPriority — добавить в очередь элемент с назначенным приоритетом;
- Dequeue — извлечь элемент с наибольшим (наименьшим) приоритетом;

- Peek — получить значение элемента с наибольшим (наименьшим) приоритетом без удаления.

Другими словами, в очереди хранятся пары вида (ключ, значение), причем ключи могут быть одинаковыми. Пример такой очереди — список задач, где необходимо каждый раз браться за наиболее важную. В самой простой реализации операции будут выполняться за  $O(n)$ .

### 3.7. Граф

Графом называется непустое множество вершин и множество связей между парами вершин — ребер. Обозначается граф как  $G = (V, E)$ , где  $V$  — вершина, или vertex, а  $E$  — ребро, или edge. Обычно, вершины обозначаются окружностями, а ребра — линиями.

Ребра бывают ориентированными и неориентированными, а также взвешенными и невзвешенными. Взвешенные ребра имеют дополнительную характеристику, называемую весом или ценой пути. Вес может быть как положительным, так и отрицательным. Ребра в невзвешенном графе эквивалентны ребрам с одинаковым весом, например, с весом 1. Графы классифицируются в том числе и в зависимости от того, какие ребра они содержат, поскольку ребра в одном графе однотипны.

Граф как на рис. 3.9. называется ориентированным и обозначается как  $G = (V, A)$ , где  $A$  — дуга, или arc. Дуга представляет собой такую же пару вершин, как и ребро, но движение по ней может осуществляться только от первой вершины ко второй. Неориентированное ребро можно представить в виде двух противоположно направленных ориентированных ребер.

Цепью называется последовательность ребер, соединяющая две вершины в графе. В ориентированном графе такую последовательность еще называют путем. Одна из основных задач в графе — минимизация путей между вершинами. Путь из вершины в нее саму называется циклом.



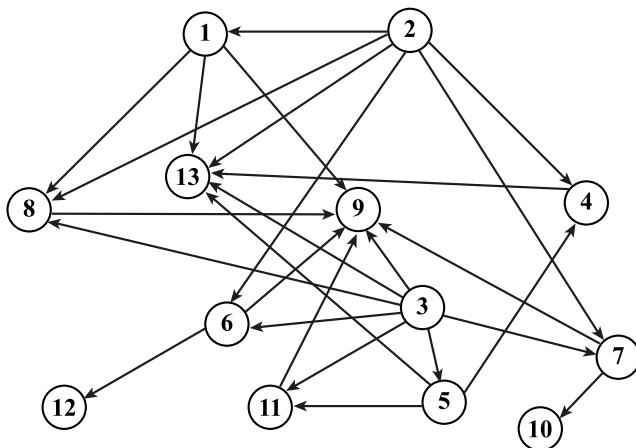


Рис. 3.8. Ориентированный граф

У графа существует два стандартных представления: в виде списка смежных вершин и в виде матрицы смежности.

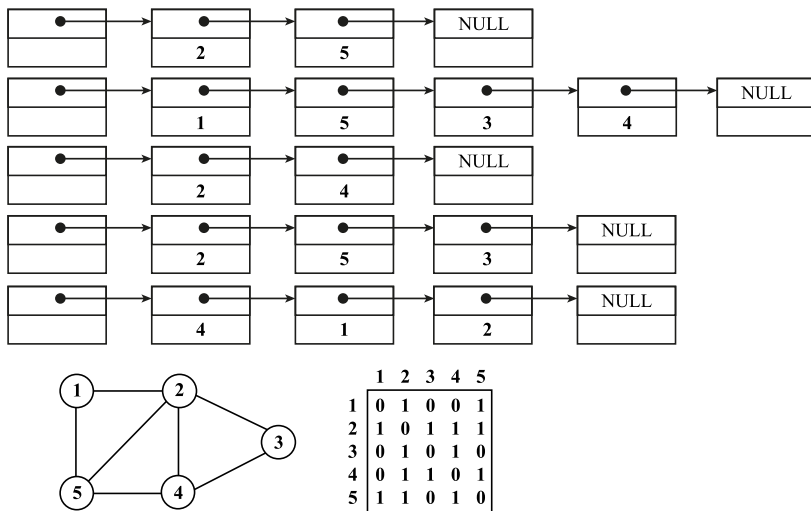


Рис. 3.9. Представление графов

Граф называется полным в случае, когда из каждой его вершины есть ребро в каждую другую вершину. Для графа, имеющего  $n$  вершин, таких ребер будет  $\frac{n(n-1)}{2}$ .

Представление графа  $G = (V, E)$  в виде списка смежности (англ. adjacency list representation) использует массив из  $|V|$  списков, по одному для каждой вершины из  $V$ . Для каждой вершины список содержит все вершины  $v$ , такие что  $(u, v) \in E$ , т. е. список состоит из всех вершин, смежных с  $u$  в графе  $G$  (список может содержать не сами вершины, а указатели на них). Вершины в каждом списке обычно хранятся в произвольном порядке.

Представление графа с помощью матрицы смежности предполагает, что вершины пронумерованы числами  $1, 2, \dots, |V|$  и образуют матрицу  $A$  размером  $|V| \times |V|$ , для которой

$$A_{ij} = \begin{cases} 1, & \text{если } (i, j) \in E, \\ 0, & \text{если нет} \end{cases}$$

Очевидно, что такая матрица требует объем памяти, равный  $O(V^2)$ , независимо от количества ребер графа. Значения в матрице могут задавать и веса графа. Тогда отсутствие ребра отмечается не нулем, а некоторым символом.

Обычно более предпочтительно представление с помощью списков смежности, поскольку оно обеспечивает компактное представление разреженных (англ. sparse) графов, т. е. таких, для которых  $|E|$  гораздо меньше  $|V^2|$ . Также списки смежности, в отличие от матриц смежности, позволяют задавать больше одного ребра между двумя вершинами. Большинство алгоритмов предполагают, что входной граф представлен именно в виде списка смежности. Представление при помощи матрицы смежности предпочтительнее в случае плотных (англ. dense) графов, т. е. когда значение  $|E|$  близко к  $|V^2|$  или когда нам надо иметь возможность быстро определить, имеется ли ребро, соединяющие две данные вершины. Например, алгоритмы поиска кратчайших путей для всех пар вершин используют представление графов именно в виде матриц смежности.

Списки смежности легко адаптируются для представления взвешенных графов. Вес  $w(u, v)$ , где  $u$  и  $v$  — пара вершин, которые соединены ребром, хранится вместе с вершиной  $v$  в списке смежности  $u$ .

Недостатком представления графа в виде списка смежности является то, что проверить наличие в графе ребра  $(u, v)$  можно только прямым поиском в списках.

Для задания ребра в списке воспользуемся структурой node:

```
typedef struct {  
    int v; node* next;  
} node;
```

В случае взвешенного графа достаточно добавить еще один параметр в эту структуру — weight. Параметр v, указывающий номер вершины, и параметр weight должны быть определены при создании.

Допустим, в неориентированном графе есть 5 вершин, и мы хотим задать ребра из вершины 1 во все остальные. Тогда код для создания такого списка смежности будет следующим:

```
node* list [5];  
for (int i = 0; i < 5; ++i) list [i] = 0;  
for (int i = 1; i < 5; ++i) {  
    node* new_node = (node*) malloc (sizeof (node));  
    new_node->v = i; new_node->next = list [0];  
    list [0] = new_node;  
    list [i] = (node*) malloc (sizeof (node));  
    list [i] ->v = 0; list [i] ->next = 0;  
}
```

В итоге в первом списке будет значиться 4 записи, а в остальных — по одной. Массив из указателей на node необходимо инициализировать перед использованием. Важно: после окончания работы необходимо очистить память от каждого элемента списка с помощью команды free. Например, вот так:

```
for (int i = 0; i < 5; ++i) {  
    node* cur = list [i];  
    while (cur->next != 0 && cur->next->next != 0) {  
        node* buf = cur;  
        cur = cur->next;  
        free (buf);  
    }  
    if (cur->next == 0) free (cur);  
    else if (cur->next->next == 0) {  
        free (cur->next);  
    }  
}
```

```
        free (cur);  
    }  
}
```

Матрицу смежности графа обычно задают двумерным массивом:

```
int adjacencyMatrix [10] [10];
```

Матрица является симметричной в случае, когда граф неориентированный, и произвольной в ином случае. В качестве матрицы неориентированного графа можно также использовать верхне- или нижнетреугольную матрицу, то есть такую, которая ниже и выше главной диагонали соответственно заполнена нулями, и рассматривать только заполненную часть.

Граф, в котором все вершины разбиты на два подмножества, а каждое ребро соединяет вершины из разных подмножеств, называется двудольным (рис. 3.10). На взвешенных двудольных графах тривиальной задачей является задача нахождения максимального паросочетания, то есть такого набора ребер, в котором ни одно из них не имеет общей вершины с другим, и при этом ребра набора имеют максимальный вес. Задачу нахождения максимального паросочетания можно расширить и на произвольные графы.

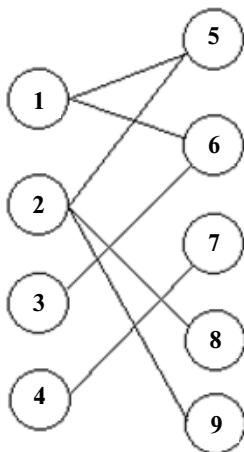


Рис. 3.10. Двудольный граф

В графе существует такое понятие, как компонента связности, то есть множество вершин графа, для каждой из которых есть путь до остальных. Каждую компоненту связности можно рассматривать как отдельный граф, со своими списками смежности. Если соединить вершину из одной компоненты связности с вершиной из другой компоненты связности ребром, и это будет единственное ребро между этими двумя компонентами, то такое ребро будет называться мостом. То есть при удалении моста количество компонент связности увеличивается на 1. Поэтому в теории графов существует задача нахождения мостов.

Если компонента связности всего одна, то граф называется связным, то есть для любой пары вершин  $(u, v)$  существует путь из  $u$  в  $v$ . Если в связном графе нет циклов (путей длины 1 и больше, которые начинаются и заканчиваются в одной вершине), то такой граф называется деревом.

### 3.8. *Дерево*

Для дерева определены следующие понятия:

- Корень — вершина, в которую не входит ни одного ребра.
- Дерево с отмеченной вершиной называется корневым деревом.
- Степень узла — количество исходящих дуг (или иначе количество поддеревьев узла).
- Концевой узел (лист, терминальная вершина) — узел со степенью 1 (то есть узел, в который ведет только одно ребро; в случае ориентированного дерева — узел, в который ведет только одна дуга и из которого не исходит ни одной дуги).
- Узел ветвления — не концевой узел.
- Уровень узла — длина пути от корня до узла. Можно определить рекурсивно: уровень корня дерева равен 0, уровень любого другого узла на единицу больше, чем уровень корня ближайшего поддерева дерева, содержащего данный узел.

Бинарным (двоичным) называется такое дерево, в котором каждый узел имеет не более двух потомков. Как правило, узел называется родительским, а потомки — левым и правым наследниками.

Двоичные деревья упорядочены, то есть различают левое и правое поддерева. Типичным примером двоичного дерева является генеа-

логическое дерево (родословная). В других случаях двоичные деревья используются тогда, когда на каждом этапе некоторого процесса надо принять одно решение из двух возможных.

Строго двоичным деревом называется дерево, у которого каждая внутренняя вершина имеет непустые левое и правое поддеревья. Это означает, что в строго двоичном дереве нет вершин, у которых есть только одно поддерево.

Полным двоичным деревом называется дерево, у которого все листья находятся на одном уровне и каждая внутренняя вершина имеет непустые левое и правое поддеревья.

Пример дерева приведен на рис. 3.11.

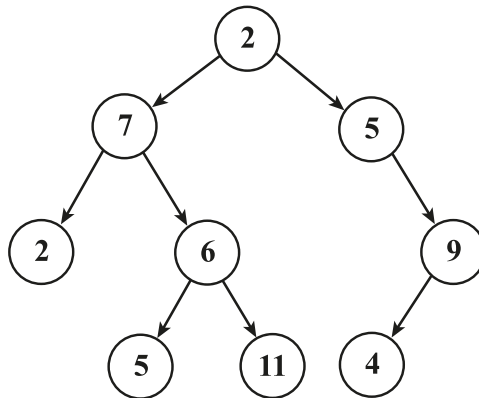


Рис. 3.11. Дерево

Существует два основных способа представления дерева — в виде разветвленного списка и в виде массива.

Для представления дерева в виде списка определим следующую структуру узла:

```
struct node {  
    int elem;  
    node* left;  
    node* right;  
    node* parent;  
};
```

Внутри структуры объявлены два указателя на такие же структуры — сыновья узла, и один указатель на структуру родителя. Выглядеть это представление дерева будет так, как показано на рис. 3.12.

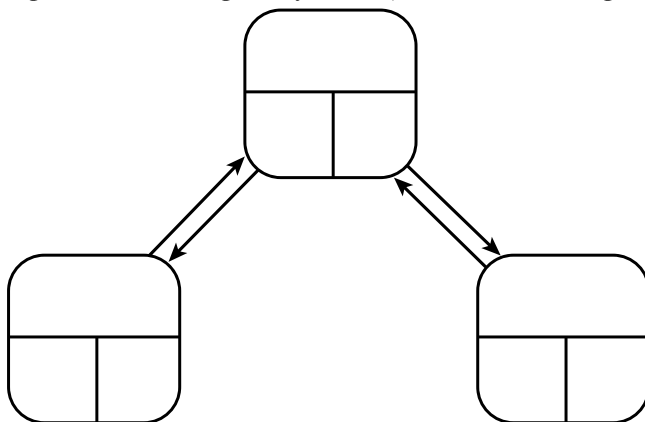


Рис. 3.12. Представление дерева в виде списка

Такое представление удобно для двоичных деревьев, а также для деревьев, в которых количество детей узла (степень узла) фиксировано. Если степени узлов сильно отличаются и принимают значения в промежутке от 0 до  $n$ , то тогда имеет смысл хранить в узле переменную количества детей и указатель на список (или массив) детей. Пример такой структуры приведен ниже.

```

struct node {
    int elem;
    int child_count;
    node* parent;
    node* child_list;
};
  
```

Здесь `child_list` указывает на фиктивное начало списка детей, `child_count` хранит количество детей, а `parent` указывает на родителя.

Другой способ представления деревьев организуется на массиве. Для каждого элемента рассчитывается, где будут лежать его потомки. В случае бинарного дерева для элемента  $i$  потомки хранятся в ячейках с номерами  $2i + 1$  и  $2i + 2$  (корень — в нулевой ячейке). Для дере-

ва, в котором у узла может быть 3 потомка, для элемента  $i$  они будут храниться в ячейках  $3i$ ,  $3i + 1$ ,  $3i + 2$ . И так далее. Пример хранения бинарного дерева в массиве приведен на рис. 3.13.

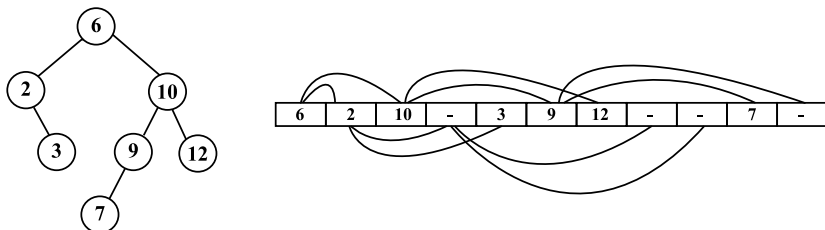


Рис. 3.13. Представление бинарного дерева в виде массива

Как можно увидеть из рисунка, недостатком такого способа является выделение дополнительной памяти под потомков вне зависимости от того, есть они или нет.



#### 4. АЛГОРИТМЫ СОРТИРОВКИ

---

**П**остановка задачи. Пусть имеется последовательность  $(a_1, a_2, \dots, a_n)$ .

Определим задачу сортировки: необходимо изменить порядок  $(a_1, a_2, \dots, a_n)$  на  $(a'_1, a'_2, \dots, a'_n)$  так, что  $a'_1 \leq a'_2 \leq \dots \leq a'_n$  (или  $\geq$ ).

Алгоритмы сортировки имеют большое практическое применение. Их можно встретить там, где речь идет об обработке и хранении больших объемов информации. Если данные заранее упорядочить, то задачи обработки данных решаются проще.

Сортировка является одной из фундаментальных алгоритмических задач программирования. Решению проблем, связанных с сортировкой, посвящено множество научных исследований, разработано множество алгоритмов. В процессе реализации алгоритмов сортировки на передний план выходят многие прикладные проблемы. Выбор наиболее производительной программы сортировки в той или иной ситуации может зависеть от многих факторов, таких как предварительные знания о ключах и сопутствующих данных, об иерархической организации памяти компьютера (наличии кэша и виртуальной памяти) и программной среды.

Сортировать можно массивы, связанные списки, строки (в лексикографическом порядке). На практике сортируемые значения редко являются изолированными, обычно это *записи* (record), а сортируются *ключи* (key). При сортировке ключей выполняется перестановка сопутствующих ключу данных, и если эти данные имеют значительный объем, то желательно проводить сортировку массива указателей на данные.

Оценка алгоритмов сортировки производится по следующим параметрам.

Время сортировки. Это основной параметр, характеризующий быстродействие алгоритма. Хорошей мерой эффективности могут служить:

- число необходимых сравнений ключей;
- число перестановок элементов.

Эти характеристики являются функциями от  $n$  — количества сортируемых элементов.

**Память.** Ряд алгоритмов требует выделения дополнительной памяти под временное хранение данных. При оценке используемой памяти не будет учитываться место, которое занимает исходный массив и независимые от входной последовательности затраты, например на хранение кода программы.

**Устойчивость.** Метод сортировки называется устойчивым, если в процессе сортировки не меняется относительное расположение элементов с равными ключами.

**Естественность поведения.** Эффективность метода при обработке уже отсортированных или частично отсортированных данных. Алгоритм ведет себя естественно, если учитывает эту характеристику входной последовательности.

Сортировки делятся на внутренние и внешние.

Внутренние сортировки используются для данных, которые можно поместить в оперативную память ЭВМ.

Внешние сортировки упорядочивают информацию, расположенную на внешних носителях.

Это накладывает некоторые дополнительные ограничения на алгоритм:

- доступ к носителю осуществляется последовательным образом: в каждый момент времени можно считать или записать только один элемент;
- объем данных не позволяет им разместиться в ОЗУ;
- доступ к данным на носителе производится намного медленнее, чем операции с оперативной памятью.

В базах данных в большинстве случаев используются внешние сортировки, более того, данные могут располагаться на разных устройствах, соединенных локальной сетью или сетью Интернет.

Рассмотрим прямые методы сортировки. Прямые методы сортировки в соответствии с принципами, положенными в их основу, делятся на три категории:

- сортировки выбором;
- сортировки обментами;
- сортировки включением (вставками).

### 4.1. Сортировка выбором

Сортировка выбором состоит из следующих шагов. На первом шаге выбираем наименьший из элементов  $a_1 \dots a_n$  и меняем его местами с  $a_1$ . На  $i$ -ом шаге выбираем наименьший из элементов  $a_i \dots a_n$  и меняем его местами с  $a_i$ . На шаге с номером  $(n - 1)$  выбираем наименьший из элементов  $a_{n-1}$  и  $a_n$  и меняем его местами с  $a_{n-1}$ .

Рассмотрим пример сортировки массива простым выбором.

Начальное состояние массива							
6	19	3	8	92	15	1	9
Ход сортировки							
<b>6</b>	19	3	8	92	15	<b>1</b>	9
1	<b>19</b>	<b>3</b>	8	92	15	6	9
1	3	<b>19</b>	8	92	15	<b>6</b>	9
1	3	6	<b>8</b>	92	15	19	9
1	3	6	8	<b>92</b>	15	19	<b>9</b>
1	3	6	8	9	<b>15</b>	19	92
1	3	6	8	9	15	<b>19</b>	92
<b>1</b>	<b>3</b>	<b>6</b>	<b>8</b>	<b>9</b>	<b>15</b>	<b>19</b>	<b>92</b>

Рис. 4.1. Пример сортировки выбором

На первом шаге алгоритм производит  $(n - 1)$  сравнений, на последнем шаге 1 сравнение. Количество шагов равно  $(n - 1)$ . Тогда по формуле суммы первых  $n$  членов арифметической прогрессии получим

$$\frac{((n-1)+1) \cdot (n-1)}{2} = \frac{n \cdot (n-1)}{2} \text{ сравнений. В худшем случае}$$

на каждом шаге совершается 1 пересылка, соответственно количество пересылок равно  $n - 1$ . Итоговая сложность алгоритма  $O(n^2)$ .

Метод может быть неустойчивым. Пусть имеется последовательность из трех элементов, каждый из которых имеет два поля, а сортировка идет по первому из них, например  $(2a \ 2b \ 1a)$ . Здесь первое поле — ключ, второе — информационное. Работа алгоритма:  $(2a \ 2b \ 1a, 1a \ 2b \ 2a, 1a \ 2b \ 2a)$ .

Результат сортировки можно увидеть уже после шага 1, так как больше обменов не будет. Порядок ключей  $2a, 2b$  был изменен на  $2b, 2a$ , так что метод неустойчив. Если входная последовательность почти упорядочена, то сравнений будет столько же, как и в случае неупорядоченной последовательности, значит, алгоритм ведет себя неестественно.

## 4.2. Обменная сортировка

Наиболее простая для понимания сортировка, называемая также «пузырьковой сортировкой», или bubble sort.

Для последовательности  $a_1, a_2, \dots, a_n$  работает следующим образом. Начиная с конца последовательности, сравниваются два соседних элемента  $a_n$  и  $a_{n-1}$ . Если выполняется условие  $a_{n-1} > a_n$ , то значения элементов меняются местами. Процесс продолжается для  $a_{n-1}$  и  $a_{n-2}$  и т. д., пока не будет произведено сравнение  $a_2$  и  $a_1$ . После этого на месте  $a_1$  окажется элемент массива с наименьшим значением. На втором шаге процесс повторяется, последними сравниваются элементы  $a_3$  и  $a_2$ . И так далее. На последнем шаге будут сравниваться только текущие значения  $a_n$  и  $a_{n-1}$ . Хорошо просматривается аналогия с пузырьком, поскольку наименьшие элементы (самые «легкие») постепенно «всплывают» к верхней границе массива. Рассмотрим пример сортировки массива.

Начальное состояние массива							
6	19	3	8	92	15	1	9
Ход сортировки							
<b>1</b>	6	19	3	8	92	15	9
1	<b>3</b>	6	19	8	9	92	15
1	3	<b>6</b>	8	19	9	15	92
1	3	6	<b>8</b>	9	19	15	92
1	3	6	8	<b>9</b>	15	19	92
1	3	6	8	9	<b>15</b>	19	92
1	3	6	8	9	15	<b>19</b>	92
<b>1</b>	<b>3</b>	<b>6</b>	<b>8</b>	<b>9</b>	<b>15</b>	<b>19</b>	<b>92</b>

Рис. 4.2. Пример сортировки методом пузырька

Ниже приведен текст функции пузырьковой сортировки на языке C:

```
void BubbleSort (int n, int* x) {
    int buf = 0;
    for (int i = n - 1; i >= 0; --i)
        for (int j = i - 1; j >= 0; --j)
            if (x[j] > x[j + 1]) {
                buf = x[j];
                x[j] = x[j + 1];
                x[j + 1] = buf;
            }
}
```

Как и при сортировке выбором, на первом шаге производится  $(n - 1)$  сравнений, на последнем 1 сравнение. Всего шагов  $(n - 1)$ . Худший случай предполагает  $n/2$  пересылок в среднем на каждом шаге, то есть всего  $\frac{n * (n - 1)}{2}$  пересылок. Итоговая сложность равна  $O(n^2)$ .

Метод пузырька можно усовершенствовать:

- если на некотором шаге не было произведено ни одного обмена, то выполнение алгоритма можно прекращать;
- можно запоминать наименьшее значение индекса  $k$ , для которого на текущем шаге выполнялась последняя перестановка. Верхняя часть массива до элемента с этим индексом уже отсортирована, и на следующем шаге можно прекращать сравнения значений соседних элементов при достижении такого значения индекса, т. к. все пары соседних элементов с индексами, меньшими  $k$ , уже расположены в нужном порядке. Дальнейшие проходы можно заканчивать на индексе  $k$  вместо того, чтобы двигаться до установленной заранее верхней границы  $i$ ;
- метод пузырька работает неравноправно для «легких» и «тяжелых» значений. Минимальный элемент за один цикл сдвигается на свое место, а максимальный — всего лишь на одну позицию. Так что массив (23451) будет отсортирован за 1 проход, а сортировка массива (51234) потребует 4 прохода. Для улучшения работы алгоритма можно менять направление следующих один за другим проходов. Получившийся алгоритм называют шейкер-сортировкой (cocktail sort или shaker sort). При его применении на каждом следующем шаге меняется направление последовательного просмотра. В результате на одном шаге «всплывает» очередной наиболее легкий элемент, а на другом — «тонет» очередной самый тяжелый.

Общая асимптотика решения не улучшается, но на отсортированном массиве количество сравнений равно  $(n - 1)$ , а асимптотика, соответственно, равна  $O(n)$ .

Ниже приведен текст функции шейкерной сортировки на языке C:

```
void ShakerSort (int n, int* x)
    int Left = 0, Right = n - 1, buf = 0;
    do {
        for (int i = Right; i >= Left; —i) {
```

```
        if (x [i - 1] > x [i]) {  
            buf = x [i];  
            x [i] = x [i - 1];  
            x [i - 1] = buf;  
        }  
    }  
    Left = Left + 1;  
    for (int i = Left; i <= Right; ++i) {  
        if (x [i - 1] > x [i]) {  
            buf = x [i];  
            x [i] = x [i - 1];  
            x [i - 1] = buf;  
        }  
    }  
    Right = Right - 1;  
}  
while (Left <= Right);  
}
```

В данном случае границы области сортировки обозначаются двумя переменными Left и Right. Соответственно, после первого цикла наименьший элемент занимает позицию  $x[\text{Left}]$ , а после прохождения второго цикла наибольший элемент занимает позицию  $x[\text{Right}]$ .

Поведение метода шейкерной сортировки в отличие от строгого метода прямых обменов довольно естественное, почти отсортированный массив будет отсортирован намного быстрее случайного. С другой стороны, строгий метод обменной сортировки является устойчивым, в то время как шейкерная сортировка этим качеством не обладает. Рассмотренные сортировки обменом, даже с улучшениями, не имеют практической ценности в силу своей низкой эффективности.

### *4.3. Сортировка вставкой*

В сортировке пузырьком или выбором можно было четко заявить, что на  $i$ -м шаге элементы  $a_1, a_2, \dots, a_i$  стоят на правильных местах и никуда более не переместятся. В рассматриваемом алгоритме подобное утверждение будет более слабым: последовательность  $a_1, a_2, \dots, a_i$  упо-

рядочена. При этом по ходу алгоритма в нее будут вставляться новые элементы.

Пусть имеется массив ключей  $a_1, a_2, \dots, a_n$ . Для каждого элемента массива, начиная со второго, производится сравнение с элементами с меньшим индексом: элемент  $a_i$  последовательно сравнивается с элементами  $a_{i-1}$ ,  $a_{i-2}$  ... до тех пор, пока для очередного элемента  $a_j$  не выполнится соотношение  $a_j > a_i$ , тогда  $a_i$  и  $a_j$  меняются местами. Проверка продолжается дальше.

Переход к обработке следующего элемента  $a_{i+1}$  будет производиться в том случае, если удастся встретить такой элемент  $a_j$ , что  $a_j \leq a_i$ , или если достигнута нижняя граница массива. Алгоритм завершается, если обработана верхняя граница массива. Рассмотрим пример сортировки вставкой.

Начальное состояние массива							
6	19	3	8	92	15	1	9
Ход сортировки							
<b>6</b>	19	3	8	92	15	1	9
6	<b>19</b>	3	8	92	15	1	9
<b>3</b>	6	<b>19</b>	8	92	15	1	9
3	6	<b>8</b>	<b>19</b>	92	15	1	9
3	6	8	19	<b>92</b>	15	1	9
3	6	8	<b>15</b>	19	<b>92</b>	1	9
<b>1</b>	3	6	8	15	19	<b>92</b>	9
<b>1</b>	<b>3</b>	<b>6</b>	<b>8</b>	<b>9</b>	<b>15</b>	<b>19</b>	<b>92</b>

Рис. 4.3. Пример сортировки вставкой

Ниже приведен текст функции сортировки вставкой на языке C:

```
void InsertionSort (int n, int* x) {
    for (int i = 1; i < n; ++i) {
        int j = i - 1, key = x [i];
        while (j >= 0 && x [j] > key) {
            x [j + 1] = x [j];
            --j;
        }
        x [j + 1] = key;
    }
}
```

На каждом шаге алгоритма выполняется в среднем  $n/2$  сравнений. В лучшем случае (массив отсортирован) пересылки выполняться

не будут. В худшем случае (массив отсортирован в обратном порядке) на каждом шаге в среднем будет выполнено  $n/2$  пересылок. Количество шагов алгоритма равно  $(n - 1)$ . Тогда общее количество пересылок в худшем случае равно  $\frac{n * (n - 1)}{2}$ , количество сравнений также равно  $\frac{n * (n - 1)}{2}$ . Общая сложность алгоритма  $O(n^2)$ .

Хорошим показателем сортировки является весьма естественное поведение: почти отсортированный массив будет досортирован очень быстро. Это вместе с устойчивостью алгоритма делает метод хорошим выбором при относительно невысоких значениях  $n$ .

Сортировка вставкой хорошо применима для сортировки связанных списков (описаны в разделе 3.4). Каждый следующий элемент  $i$  извлекается из списка и устанавливается на свое место в отсортированном массиве  $[1..i - 1]$ . Пример сортировки списка изображен на рис. 4.4.

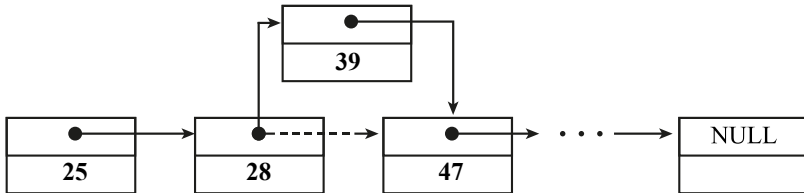


Рис. 4.4. Сортировка связанного списка методом вставки

Рисунок отражает один шаг преобразования сортировки связанного списка. Элемент со значением 39 вставляется на свое место в отсортированной части списка, состоящей из элементов 25, 28 и 47. Изменения в структуре списка при этом такие же, как и при добавлении нового элемента в произвольную часть списка.

#### 4.4. Сортировка слиянием

Алгоритм внутренней сортировки со слияниями (merge sort) — один из популярных алгоритмов сортировки. Использует метод «разделяй и властвуй». Основная идея алгоритма заключается в методе слияния двух отсортированных последовательностей. Этот же принцип является основой внешней сортировки.



Пусть имеются два отсортированных в порядке возрастания массива с элементами  $A_1, A_2, \dots, A_n$  и  $B_1, B_2, \dots, B_n$  и имеется пустой массив  $C_1, C_2, \dots, C_{2n}$ , который мы хотим заполнить значениями массивов  $A$  и  $B$  в порядке возрастания.

Для слияния выполняются следующие действия: сравниваются  $A_1$  и  $B_1$ , и меньшее из значений записывается в  $C_1$ . Предположим, что это значение  $A_1$ . Тогда  $A_2$  сравнивается с  $B_1$ , и меньшее из значений заносится в  $C_2$ . Предположим, что это значение  $B_1$ . Тогда на следующем шаге сравниваются значения  $A_2$  и  $B_2$  и т. д., пока мы не достигнем границ одного из массивов. Остаток другого массива просто дописывается в «хвост» массива  $C$ .

Для сортировки со слиянием исходного неупорядоченного массива  $a_1, a_2, \dots, a_n$  заводится парный массив  $b_1, b_2, \dots, b_n$ .

На первом шаге производится слияние  $a_1$  и  $a_n$  с размещением результата в  $b_1, b_2$ , слияние  $a_2$  и  $a_{n-1}$  с размещением результата в  $b_3, b_4$  и так далее. Результат слияния  $a_{n/2}$  и  $a_{n/2+1}$  помещается в  $b_{n-1}, b_n$ .

На втором шаге производится слияние пар  $b_1, b_2$  и  $b_{n-1}, b_n$  с помещением результата в  $a_1, a_2, a_3, a_4$ , слияние пар  $b_3, b_4$  и  $b_{n-3}, b_{n-2}$  с помещением результата в  $a_5, a_6, a_7, a_8$ , ..., слияние пар  $b_{n/2-1}, b_{n/2}$  и  $b_{n/2+1}, b_{n/2+2}$  с помещением результата в  $a_{n-3}, a_{n-2}, a_{n-1}, a_n$ . И так далее.

На последнем шаге производится слияние последовательностей элементов массива длиной  $n/2$ :  $a_1, a_2, \dots, a_{n/2}$  и  $a_{n/2+1}, a_{n/2+2}, \dots, a_n$  с помещением результата в  $b_1, b_2, \dots, b_n$  (или наоборот в  $a_1, a_2, \dots, a_n$  в зависимости от значения  $n$ ).

Сортировка слиянием обычно реализуется рекурсивно. На каждом шаге рекурсии массив разбивается на два подмассива, каждый из которых также разбивается на два подмассива и так далее. Рекурсия достигает своего нижнего предела, когда длина сортируемой последовательности становится равной 1. В этом случае вся работа уже сделана, поскольку любую такую последовательность из одного элемента можно считать упорядоченной. Ниже приведена функция MergeSort, которая реализует этот алгоритм.

Объединение отсортированных последовательностей осуществляется с помощью вспомогательной процедуры Merge ( $A, p, q, r$ ), где  $A$  — массив, а  $p, q$  и  $r$  — индексы, нумерующие элементы массива, такие, что  $p \leq q < r$ . В этой процедуре предполагается, что элементы подмассивов  $A[p \dots q]$  и  $A[q+1 \dots r]$  упорядочены. Она сливает эти два подмассива в один отсортированный, элементы которого заменяют текущие элементы подмассива  $A[p \dots r]$

Начальное состояние массива							
6	19	3	8	92	15	1	9
Ход сортировки							
6	19	3	8	92	15	1	9
6	19	3	8	92	15	1	9
6	19	3	8	92	15	1	9
6	19	3	8	15	92	1	9
3	6	8	19	1	9	15	92
<b>1</b>	<b>3</b>	<b>6</b>	<b>8</b>	<b>9</b>	<b>15</b>	<b>19</b>	<b>92</b>

Рис. 4.5. Пример сортировки слиянием

Код основной функции сортировки:

```
void MergeSort (int* x, int first, int last) {
    if (first < last) {
        int split = (first + last) / 2;
        MergeSort (x, first, split);
        MergeSort (x, split + 1, last);
        Merge (x, first, split, last);
    }
}
```

При длине массива большей, чем 1, рассчитывается ее середина (переменная split), а затем запускаются два рекурсивных вызова для левой и правой частей, которые при возврате управления в этот экземпляр функции соединяются в функции Merge:

```
void Merge (int* x, int first, int split, int last) {
    int pos1 = first, pos2 = split + 1, pos3 = 0;
    int* temp = new int [last - first + 1];
    while (pos1 <= split && pos2 <= last) {
        if (x [pos1] < x [pos2])
            temp [pos3++] = x [pos1++];
        else
            temp [pos3++] = x [pos2++];
    }
    while (pos2 <= last)
        temp [pos3++] = x [pos2++];
    while (pos1 <= split)
        temp [pos3++] = x [pos1++];
    for (int i = 0; i < last - first + 1; ++i)
        x [first + i] = temp [i];
}
```

Оценим сложность алгоритма. Вызов любой рекурсивной функции выглядит как дерево (раздел 3.7). Такое дерево еще называют деревом рекурсии. Каждый уровень дерева представляет собой один или несколько вызовов функции. Узлы дерева, не имеющие потомков (терминальные листья), представляют те вызовы функции, которые прекращают рекурсию. Высота дерева в случае сортировки слиянием равна  $\log_2 n$ , поскольку на каждом шаге исходный массив разбивается на два подмассива длиной  $n/2$ . После разбиения производится операция слияния. Операция слияния требует  $n$  сравнений и повторяется, соответственно,  $\log n$  раз, то есть 1 раз для каждого уровня дерева. Асимптотика решения тогда равна  $O(n \log n)$ .

#### 4.5. Быстрая сортировка

Быстрая сортировка, или quick sort, — алгоритм, также основанный на принципе «разделяй и властвуй», но при этом являющийся улучшением обменной сортировки, а также одним из самых эффективных алгоритмов в этой сфере. Основная идея алгоритма состоит в том, что случайным образом выбирается некоторый элемент  $x$  (который называется *опорным элементом*), после чего массив просматривается слева, пока не встретится элемент  $a_i > x$ , а затем массив просматривается справа, пока не встретится элемент  $a_j < x$ . Эти два элемента меняются местами, затем процесс повторяется до тех пор, пока  $i < j$ . Когда  $i$  станет равен  $j$ , массив окажется разбитым на две части — левую, в которой значения ключей будут меньше  $x$ , и правую, со значениями ключей больше  $x$ . Опишем действия, которые выполняются алгоритмом на одном шаге.

1. Пусть имеется массив  $a_1, a_2, \dots, a_n$  и опорный элемент  $x$ , по которому будет производиться разделение.
2. Введем два индекса:  $i$  и  $j$ . В начале алгоритма они указывают соответственно на левый и правый конец последовательности.
3. Будем двигать индекс  $i$  с шагом в один элемент по направлению к концу массива, пока не будет найден элемент  $a_i > x$ .
4. Аналогично индекс  $j$  последовательно уменьшается по направлению к концу массива до тех пор, пока  $j$ -й элемент не окажется меньше опорного:  $a_j < x$ .
5. Далее, если  $i < j$ , меняем  $a_i$  и  $a_j$  местами и продолжаем двигать  $i, j$  по тем же правилам с тех значений  $i$  и  $j$ , которые были достигнуты.

6. Повторяем шаги с 3-го, пока  $i < j$ .
7. Если  $i = j$  — операция разделения закончена.

Далее алгоритм рекурсивно вызывается для левой и правой частей до тех пор, пока в каждой части не окажется по 1 элементу. В отличие от сортировки слиянием, где есть фаза объединения подмассивов, в этом алгоритме по завершении рекурсии массив уже будет отсортирован.

Количество сравнений на каждом шаге равно  $(n - 1)$ . В отличие от сортировки слиянием, массив может разбиваться не на равные отрезки. Соответственно высота дерева рекурсии для быстрой сортировки может отличаться от  $\log_2 n$ . В наилучшем случае, когда опорный элемент равен среднему элементу массива, глубина рекурсии представляет собой  $\log_2 n$ , а асимптотикой будет  $O(n \log n)$ . Худшим случаем будет тот, когда опорным элементом оказывается минимальный элемент, то есть глубина рекурсии примерно будет равна  $n$ . При этом опорный элемент должен оказываться минимальным на каждом шаге, что маловероятно. В такой ситуации асимптотикой решения будет  $O(n^2)$ .

В целом алгоритм не зря называется так — большинство программ пользуются именно его вариациями. Так, стандартная библиотека C++ поддерживает функцию `qsort`, в качестве основы использующую быструю сортировку.

Код функции быстрой сортировки:

```
void QuickSort (int* x, int left, int right) {
    int i = left, j = right, buf, pivot = x [(left + right) / 2];
    do {
        while (x [i] < pivot) ++i;
        while (x [j] > pivot) --j;
        if (i < j) {
            buf = x [i]; x [i] = x [j]; x [j] = buf;
            i++; j--;
        }
    } while (i < j);
    if (j > left) QuickSort (x, left, j);
    if (i < right) QuickSort (x, i, right);
}
```

Опорный элемент — не обязательно элемент массива. Например, в качестве опорного можно выбрать среднее арифметическое началь-

ного и конечного элементов подмассива, полученного в результате рекурсии, или, например, центральный элемент подмассива. Вариантов выбора может быть довольно много, лучшим случаем, очевидно, будет тот, где опорный элемент — медиана массива, но для ее выбора нужны дополнительные действия.

Рассмотрим пример работы алгоритма быстрой сортировки (в качестве опорного выбирается центральный элемент подмассива):

Начальное состояние массива

6	19	3	8	92	15	1	9
Ход сортировки							
6	19	3	<b>8</b>	92	15	1	9
6	<b>1</b>	3	8	92	<b>15</b>	19	9
1	<b>6</b>	3	8	9	15	19	92
1	3	<b>6</b>	<b>8</b>	<b>9</b>	<b>15</b>	<b>19</b>	<b>92</b>

Рис. 4.6. Пример быстрой сортировки

Подмассив (6, 1, 3) представляет из себя худший случай, поскольку в качестве опорного элемента выбирается наименьший элемент подмассива. В итоге он просто перемещается в начало массива, меняясь местами с первым элементом.

#### 4.6. Пирамидальная сортировка

Пирамидальная сортировка, или heap sort, — алгоритм, в котором для управления информацией в ходе сортировки используется специальная структура — heap (пирамида).

Время работы этого алгоритма, как и время работы алгоритма сортировки слиянием (и в отличие от времени работы алгоритма сортировки вставкой) —  $O(n \log n)$ . Как и сортировка методом вставок, и в отличие от сортировки слиянием, пирамидальная сортировка выполняется без привлечения дополнительной памяти: в любой момент времени требуется память для хранения вне массива только некоторого постоянного количества элементов. Таким образом, в пирамидальной сортировке сочетаются лучшие особенности двух рассмотренных ранее алгоритмов сортировки.

Пирамида (binary heap) — это структура данных, представляющая собой объект-массив, который можно рассматривать как почти полное бинарное дерево. Каждый узел этого дерева соответствует опре-

деленному элементу массива. На всех уровнях, кроме, может быть, последнего, дерево полностью заполнено (заполненный уровень — это такой, который содержит максимально возможное количество узлов). Последний уровень заполняется слева направо до тех пор, пока в массиве не закончатся элементы.

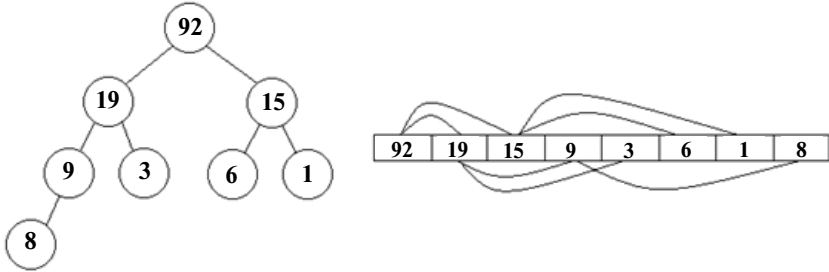


Рис. 4.7. Пирамида в виде бинарного дерева и массива

В пирамиде, представленной на рис. 4.7, число в окружности, представляющей каждый узел дерева, является значением, сортируемым в данном узле. При представлении дерева в массиве линии, попарно соединяющие элементы массива, обозначают взаимосвязь вида «родитель — потомок». Родительские элементы всегда расположены слева от сыновей. Данное дерево имеет высоту, равную 3; узел со значением 8 расположен на третьем уровне.

Различают два вида бинарных пирамид: неубывающие и невозрастающие. В пирамидах обоих видов значения, расположенные в узлах, удовлетворяют свойству пирамиды (heap property), являющемуся отличительной чертой пирамиды того или иного вида. Свойство невозрастающих пирамид (max-heap property) заключается в том, что для каждого отличного от корневого узла с индексом  $i$  выполняется следующее неравенство:

$$a[\text{parent}(i)] \geq a[i].$$

Другими словами, значение узла не превышает значение родительского по отношению к нему узла. Таким образом, в невозрастающей пирамиде самый большой элемент находится в корне дерева, а значения узлов поддерева, берущего начало в каком-то элементе, не превышают значения самого этого элемента. Принцип организации неубывающей пирамиды (min-heap) прямо противоположный. Свойство неубывающих пирамид (min-heap property) заключается в том, что для всех отличных от корневого узлов с индексом  $i$  выполняется такое неравенство:

$$a[\text{parent}(i)] \leq a[i].$$

Таким образом, наименьший элемент такой пирамиды находится в ее корне. В алгоритме пирамидальной сортировки используются невозрастающие пирамиды. Неубывающие пирамиды часто применяются в приоритетных очередях.

Рассматривая пирамиду как дерево, определим высоту ее узла как максимальную длину нисходящего пути до листьев дерева. Высота пирамиды определяется как высота ее корня. Поскольку  $n$ -элементная пирамида строится по принципу полного бинарного дерева, то ее высота равна  $O(\log n)$ . Время выполнения основных операций в пирамиде приблизительно пропорционально высоте дерева, и, таким образом, эти операции требуют для работы время  $O(\log n)$ .

Наиболее простым вариантом реализации пирамиды будет ее хранение в массиве, причем соответствие между геометрической структурой пирамиды как дерева и массивом устанавливается по следующей схеме: в  $a[0]$  хранится корень дерева, а левый и правый сыновья элемента  $a[i]$  хранятся соответственно в  $a[2i+1]$  и  $a[2i+2]$  (алгоритм описывается с учетом индексации в C).

Очевидно, что часть массива, начиная с  $\frac{n}{2}$  элемента, удовлетворяет свойству пирамиды, так как для нее индексы  $2i+1$  и  $2i+2$  находятся за границей массива. Тогда будем идти по массиву от элемента с номером  $\frac{n-1}{2}$  до элемента с номером 0 и восстанавливать свойство пирамиды:

- смотрим на сыновей слева и справа (в массиве это  $a[2i+1]$  и  $a[2i+2]$ ) и выбираем наибольшего из них;
- если этот элемент больше  $a[i]$  — меняем его с  $a[i]$  местами и повторяем шаг 1, имея в виду новое положение  $a[i]$  в массиве, иначе конец процедуры.

Реализуем этот принцип в функции BuildHeap:

```
void BuildHeap(int* x, int k, int n) {
    int new_elem = x[k], child;
    while (k <= (n-1) / 2) {
        child = 2*k+1;
        if (child < n && x[child] < x[child + 1])
            ++child;
        if (new_elem >= x[child])
            break;
        x[k] = x[child];
```

```

}
x[k] = new_elem;
}

```

Учитывая, что высота пирамиды равна  $\log n$ , BuildHeap требует  $O(\log n)$  времени. Как видно из свойств пирамиды, в корне всегда находится максимальный элемент. Отсюда вытекает алгоритм самой сортировки:

- берем верхний элемент пирамиды  $a[0] \dots a[n-1]$  (первый в массиве) и меняем с последним местами. Теперь рассматриваем массив  $a[0] \dots a[n-2]$ : для превращения его в пирамиду достаточно просеять лишь новый первый элемент;
- повторяем шаг 1, пока обрабатываемая часть массива не уменьшится до одного элемента.

Вторая фаза занимает  $O(n \log n)$  времени:  $O(n)$  раз берется максимум и происходит просеивание бывшего последнего элемента.

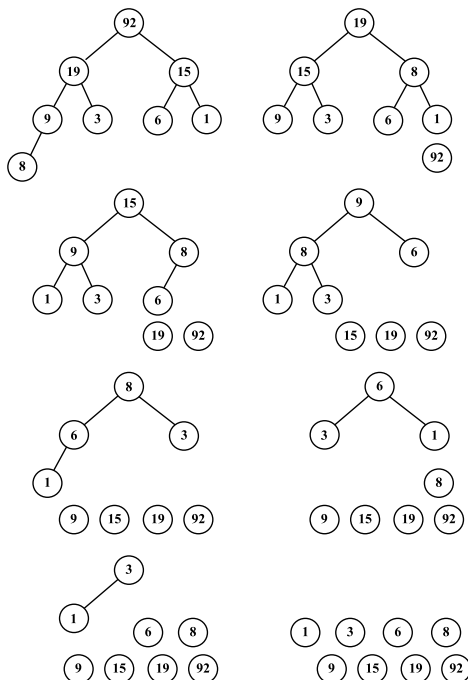


Рис. 4.8. Пример пирамидальной сортировки



Функция сортировки будет выглядеть следующим образом:

```
void HeapSort (int* x, int n) {  
    int buf;  
    for (int i = n / 2 - 1; i >= 0; --i)  
        BuildHeap (x, i, n - 1);  
    for (int i = n - 1; i > 0; --i) {  
        buf = x [i];  
        x [i] = x [0];  
        x [0] = buf;  
        BuildHeap (x, 0, i - 1);  
    }  
}
```

Функция `BuildHeap` в начале вызывается для каждого элемента из первой половины массива.

Метод является стабильным: среднее число пересылок  $\frac{n \log n}{2}$ ,

и отклонения от этого значения сравнительно малы. Пирамидальная сортировка не использует дополнительной памяти. Метод не является устойчивым: по ходу работы массив перестраивается. Поведение неестественно: частичная упорядоченность массива никак не учитывается.

#### *4.7. Внешняя сортировка*

Внешняя сортировка используется для сортировки больших объемов данных, которые не помещаются полностью в оперативную память ЭВМ. В каждый момент времени в ОП находится лишь часть полного набора. Главным критерием при разработке методов сортировки становится минимизация числа обращений к внешней памяти. Основой большинства алгоритмов внешней сортировки является принцип слияния двух упорядоченных последовательностей в единый упорядоченный набор.

Самым простым методом внешней сортировки является прямое слияние (или двухфазная сортировка прямым слиянием). Метод работает следующим образом:

1. Последовательность *A* разбивается на две половины *B* и *C*.
2. Части *B* и *C* сливаются в *A*, при этом одиночные элементы из *B* и *C* образуют упорядоченные пары.

3. Шаги 1 и 2 повторяются, при этом размер отсортированных блоков с каждой итерацией увеличивается в два раза.

Последнее слияние будет производиться для упорядоченных последовательностей размера  $n/2$ . Таким образом, метод повторяет ход сортировки слиянием. Необходимое количество проходов равно  $\log n$ , во время каждого прохода выполняется  $n$  операций при разделении и  $n$  операций при слиянии, соответственно асимптотика сортировки равна  $O(n \log n)$ .

Модификацией прямого слияния является однофазная сортировка прямым слиянием, или двухпутевая сбалансированная сортировка. При однофазной сортировке вместо слияния в одну последовательность результаты слияния сразу же распределяются по двум файлам, которые станут исходными для следующего прохода. Она требует наличия 4 файлов, по 2 входных и 2 выходных, при каждом проходе. Сортировка называется однофазной из-за того, что фазы разделения и слияния объединены в одну. Общее число пересылок при однофазной сортировке также равно  $n \log n$ .

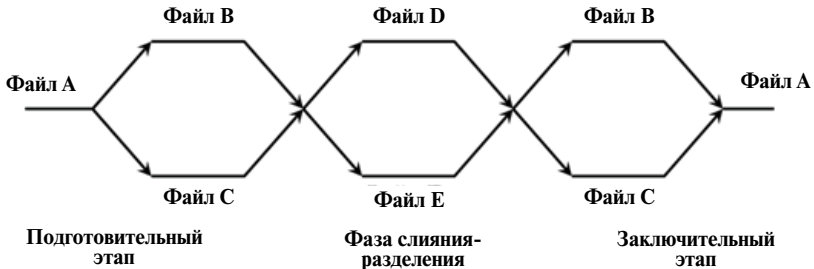


Рис. 4.9. Однофазная сортировка прямым слиянием

В этом случае после первого прохода в файлах *D* и *E* окажутся отсортированные пары, полученные из файлов *B* и *C*, из которых затем получатся отсортированные четверки элементов, и так далее.

В отличие от сортировок прямым слиянием, сортировка естественным слиянием предполагает совмещение уже отсортированных последовательностей, распределенных по нескольким входным файлам. Серией называется отсортированный набор подряд идущих элементов. Так, в массиве [1 23 42 21 25 87] две серии: [1 23 42] и [21 25 87]. Идея естественного слияния заключается в том, что серии из входного файла *A* поочередно выписываются в файлы *B* и *C*, затем первая

серия из  $B$  и первая серия из  $C$  сливаются и записываются обратно в  $A$ , потом вторая серия из  $B$  и вторая серия из  $C$ , и так далее. Если в исходном файле  $m$  серий, то в файлах  $B$  и  $C$  после одного прохода окажется  $m/2$  серий.

Если использовать четыре файла, то фазы слияния и разделения можно объединить в одну фазу. Для этого достаточно полученные при слиянии файлов  $B$  и  $C$  серии поочередно направлять в файлы  $D$  и  $E$ , и, наоборот, если сливаются серии из файлов  $D$  и  $E$ , полученные серии поочередно направлять в  $B$  и  $C$ .

Сортировка заканчивается, если при очередной фазе слияния-разделения образуется только одна серия и один из выходных файлов останется пустым, т. е. туда не будет записана ни одна серия.

Однако если входных и выходных файлов не 2, а  $N$ , то появляется возможность распределять серии элементов сразу по всем  $N$  файлам, получая  $m/N$  серий в каждом при первом проходе,  $m/N^2$  при втором,  $m/N^3$  при третьем и так далее. Такой подход называется сортировкой сбалансированным многопутевым (или  $N$ -путевым) слиянием.

Общее число проходов, необходимых для сортировки  $n$  элементов с помощью  $N$ -путевого слияния, равно  $\log_N n$ , а число пересылок при объединении фаз слияния и разделения в самом худшем случае будет равно  $n \log_N n$ . При каждом проходе будут участвовать  $N$  входных и  $N$  выходных файлов, в которые по очереди распределяются серии.

Очередной проход заканчивается, когда достигается конец всех входных файлов. Выходные файлы становятся входными, а входные файлы — выходными, и начинается очередной проход сортировки. Процесс сортировки заканчивается, когда при очередном проходе будет сформирована одна-единственная серия.

Алгоритм слияния эффективно работает на длинных сериях и неэффективно на коротких. Поэтому его нецелесообразно применять с самого начала сортировки, поскольку для случайного входного набора на первых шагах будут получаться очень короткие серии, что приведет к неоправданно большому числу обращений к памяти.

Для устранения этого недостатка можно поступить следующим образом:

1. Исходный сверхбольшой набор данных разделить на отдельные фрагменты такого размера, чтобы каждый фрагмент мог целиком разместиться в ОП.

2. Каждый фрагмент отдельно загрузить в ОП и отсортировать алгоритмом внутренней сортировки.
3. Каждый отсортированный фрагмент после этого рассматривать как серию. Полученные серии рассортировать по выходным файлам.

Эти действия носят подготовительный характер. Принцип изображен на рис. 4.10.

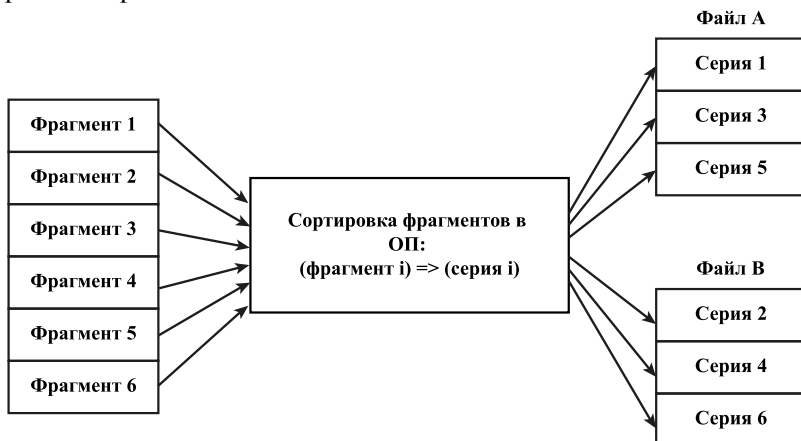


Рис. 4.10. Подготовка к внешней сортировке

К полученным файлам затем применяется один из предложенных ранее методов сортировки естественным слиянием. С каждым шагом в выходных файлах будут формироваться серии в два раза длиннее, пока не получится одна серия длиной  $n$ .

## 5. АЛГОРИТМЫ ПОИСКА

---

Задача поиска привлекала большое внимание ученых (программистов) еще на заре компьютерной эры. С 50-х годов началось решение проблемы поиска элементов, обладающих определенным свойством в заданном множестве. Алгоритмам поиска посвятили свои труды J. von Neumann, K. E. Batchner, J. W. J. Williams, R. W. Floyd, R. Sedgewick, E. J. Isaac, C. A. R. Hoare, D. E. Knuth, R. C. Singleton, D. L. Shell и другие. Исследования алгоритмов поиска ведутся и в настоящее время.

Задачу поиска можно сформулировать так: найти один или несколько элементов в множестве, причем искомые элементы должны обладать определенным свойством. Это свойство может быть абсолютным или относительным. Относительное свойство характеризует элемент по отношению к другим элементам, например минимальный элемент в множестве чисел. Пример задачи поиска элемента с абсолютным свойством: найти в конечном множестве пронумерованных элементов элемент с номером 13, если такой существует.

### 5.1. Поиск подстроки в строке

В программах, предназначенных для редактирования текста, часто возникает задача поиска всех фрагментов текста, которые совпадают с заданным образцом. Обычно текст — это редактируемый документ, а образец — искомое слово, введенное пользователем. Эффективные алгоритмы решения этой задачи повышают скорость ответа в текстовых редакторах на подобные запросы. Кроме того, алгоритмы поиска подстрок используются, например, для поиска заданных образцов в молекулах ДНК.

При формальной постановке задачи поиска подстроки (string-matching problem) предполагается, что текст задан в виде массива

$T[1, \dots, n]$ , а образец — в виде массива  $S[1, \dots, m]$ , где  $m \leq n$ , при этом элементы массивов  $S$  и  $T$  — символы из конечного алфавита  $\Sigma$ . Алфавит может иметь вид  $\Sigma = \{0, 1\}$  или  $\Sigma = \{a, b, \dots, z\}$ . Массив  $T$  также часто называют строкой, а массив  $S$  — подстрокой.

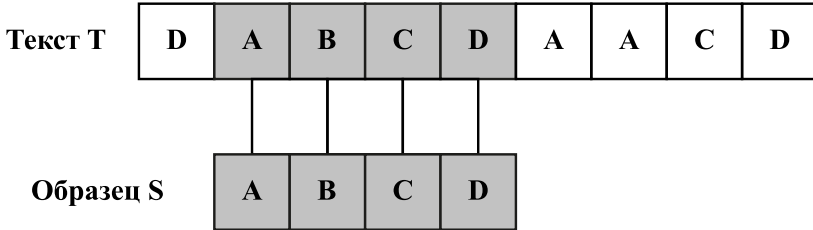


Рис. 5.1. Пример задачи поиска подстроки

Говорят, что образец  $S$  встречается в тексте  $T$  со сдвигом  $k$ , если  $T[k+1, \dots, k+m] = S[1, \dots, m]$ , где  $0 \leq k \leq n-m$ . Если образец  $S$  встречается в тексте  $T$  со сдвигом  $k$ , то величину  $k$  называют допустимым сдвигом (valid shift); в противном случае ее называют недопустимым сдвигом (invalid shift). Задача поиска подстроки — это задача поиска первого допустимого сдвига (или всех), с которыми заданный образец  $S$  встречается в тексте  $T$ .

### 5.1.1. Прямой поиск

Данный алгоритм еще называется алгоритмом последовательного поиска, он является самым простым и очевидным. Основная идея алгоритма прямого поиска заключается в посимвольном сравнении строки с подстрокой.

В начальный момент происходит сравнение первого символа строки с первым символом подстроки, второго символа строки со вторым символом подстроки и т. д. Если произошло совпадение всех символов, то фиксируется факт нахождения подстроки. В противном случае производится сдвиг подстроки на одну позицию вправо и повторяется посимвольное сравнение, то есть сравнивается второй символ строки с первым символом подстроки, третий символ строки со вторым символом подстроки и т. д. (рис. 5.2). Символы, которые сравниваются, на рисунке выделены жирным. Сдвиг подстроки повторяется до тех пор, пока конец подстроки не достиг конца строки или не произошло полное совпадение символов подстроки с символами строки.

Строка	A	B	A	B	A	B	C	A	B	A	B	C	D
Подстрока	A	B	A	B	C	D							
		A	B	A	B	C	D						
			A	B	A	B	C	D					
				A	B	A	B	C	D				
					A	B	A	B	C	D			
						A	B	A	B	C	D		
							A	B	A	B	C	D	
								A	B	A	B	C	D

Рисунок 5.2. Принцип работы алгоритма прямого поиска

Пример реализации функции поиска на языке C:

```

int NaiveStringSearch (char* t, int n, char* s, int m) {
    for (int i = 0; i <= n - m; ++i)
        for (int j = i; j <= i + m - 1; ++j)
            if (s[j - i] != t[j])
                break;
        else
            if (j == i + m - 1)
                return i;
    return -1;
}

```

Возвращаемое значение — сдвиг первого вхождения подстроки в строку, варьируется от 0 до  $(n - m)$  или равно  $-1$ , если подстрока не содержится в строке.

Данный алгоритм не нуждается в предварительной обработке и в дополнительном пространстве. Большинство сравнений алгоритма прямого поиска являются лишними. Поэтому в худшем случае алгоритм будет малоэффективен, так как его сложность будет пропорциональна  $O((n - m + 1) * m)$ , где  $n$  и  $m$  — длины строки и подстроки соответственно.

### 5.1.2. Алгоритм Рабина — Карпа

Алгоритм поиска подстроки в строке с использованием хеширования — преобразования по определенному алгоритму входных данных произвольной длины в выходные данные фиксированной длины. Функция, с помощью которой осуществляется преобразование, называется хеш-функцией.

В случае со строкой один из лучших способов определить хеш-функцию следующий:

$$h(S) = S[1] + S[2]p + S[3]p^2 + \dots + S[n]p^n,$$

где  $p$  — некоторое число.

В качестве  $p$  обычно выбирают простое число, большее количества символов во входном алфавите. Очевидно, что даже при небольшой длине строки (15–20 символов) значение хеша будет переполнять любой целочисленный тип. Однако на это не нужно обращать внимание. Фактически мы берем значение по модулю.

Пример вычисления хеша на языке C:

```
const int p = 31;
long long hash = 0, p_pow = 1;
for (size_t i = 0; i < n; ++i) {
    hash += (s[i] - «a» + 1) * p_pow;
    p_pow *= p;
}
```

Желательно отнимать «a» — 1 от кода. Таким образом, мы получаем нумерацию символов с единицы для буквенного алфавита.

Благодаря следующему свойству возможно вычисление хеша подстроки за время  $O(1)$ :

$$h[i\dots j] = S[i] + S[i+1]p + S[i+2]p^2 + \dots + S[j]p^{j-i}$$

$$h[i\dots j]p^i = S[i]p^i + S[i+1]p^{i+1} + S[i+2]p^{i+2} + \dots + S[j]p^j$$

$$h[i\dots j]p^i = h[1\dots j] - h[1\dots i-1]$$

То есть для вычисления хеша любой подстроки для строки  $S$  достаточно знать хеши всех ее префиксов. Единственной проблемой является деление на  $p^i$ , которое можно обойти следующим образом:



пусть дан хеш, умноженный на  $p^i$ , и хеш, умноженный на  $p^j$ , тогда приведем эти два хеша к одной степени, и после этого их можно сравнить<sup>4)</sup>.

Недостатком любого алгоритма, основанного на хешировании, является то, что существует вероятность совпадения хешей строк, поскольку значение хеша берется по модулю. Таким образом, разделяют истинные совпадения и ложные (spurious hit). Рассмотрим следующий пример:

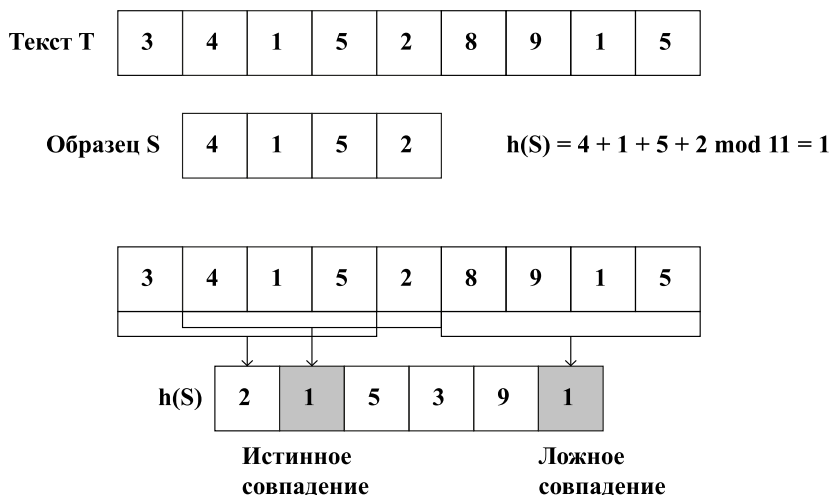


Рис. 5.3. Пример работы поиска с хешированием

Для простоты в этом примере  $p$  положено равным 1. Как можно увидеть, значение хеша 1 соответствует двум подстрокам. Поэтому при совпадении хешей необходимо посимвольное сравнение искомой подстроки с найденным вхождением.

В большинстве приложений ожидается небольшое количество вхождений подстроки (возможно, выражающееся некоторой константой  $c$ ); в таких приложениях математическое ожидание времени работы алгоритма равно  $O((n - m + 1) + cm) = O(n + m)$ . В общем случае можно ожидать, что число ложных совпадений равно  $O(n/q)$ , потому что вероятность того, что произвольное число будет эквивалентно  $p$  по модулю  $q$ , можно оценить как  $1/q$ . Поскольку имеется

<sup>4)</sup> Головешкин В. А., Ульянов М. В. Указ. соч.

всего  $O(n)$  позиций, в которых проверка на истинность совпадения дает отрицательный результат, а на обработку каждого совпадения затрачивается время  $O(m)$ , математическое ожидание времени сравнения в алгоритме Рабина — Карпа равно  $O(n) + O(m(v + q/n))$ , где  $v$  — количество допустимых сдвигов. Если  $v = O(1)$ , а  $q$  выбрано так, что  $q \geq m$ , то приведенное выше время выполнения равно  $O(n)$ . Другими словами, если математическое ожидание количества допустимых сдвигов мало ( $O(1)$ ), а выбранное простое число  $q$  превышает длину образца, то можно ожидать, что для выполнения фазы сравнения процедуре Рабина — Карпа потребуется время  $O(n + m)$ . Поскольку  $n \geq m$ , то математическое ожидание времени сравнения равно  $O(n)$ <sup>5)</sup>.

Реализация функции алгоритма Рабина — Карпа на языке C приведена ниже:

```
int RabinKarp (char* t, int n, char* s, int m) {
    const int p = 31;
    long long *p_pow = (long long*) malloc (n*sizeof (long long));
    p_pow [0] = 1;

    for (int i = 1; i < n; ++i) {
        p_pow [i] = p_pow [i-1] * p;
    }

    vector <long long> h (n);
    for (int i = 0; i < n; ++i) {
        h [i] = (t [i] - «a» + 1) * p_pow [i];
        if (i) h [i] += h [i - 1];
    }

    long long h_s = 0;
    for (int i = 0; i < m; ++i) {
        h_s += (s [i] - «a» + 1) * p_pow [i];
    }

    for (int i = 0; i + m - 1 < n; ++i) {
        long long cur_h = h [i + m - 1];
        if (i) cur_h -= h [i - 1];
```

<sup>5)</sup> Алгоритмы: построение и анализ. М., 2005.

```

        if (cur_h == h_s * p_pow[i]) {
            free(p_pow);
            return i;
        }
    }
    free(p_pow);
    return -1;
}

```

### 5.1.3. Алгоритм Кнута, Морриса и Пратта

Алгоритм основан на построении префикс-функции. Префикс-функция строки  $\pi(S, i)$  — это длина наибольшего префикса строки, который не совпадает с этой строкой и одновременно является ее суффиксом. Для строки  $S$  удобно представлять префикс-функцию в виде вектора длиной  $|S| - 1$ . Можно рассматривать префикс-функцию длины  $|S|$ , положив  $\pi(S, 1) = 0$ . Например, для строки «ababcd» префикс-функция равна  $[0, 0, 1, 2, 0, 0]$ , а для строки «cbabc» она равна  $[0, 0, 0, 1, 2, 1]$ :

Тривиальный алгоритм расчета префикс-функции работает за  $O(n^3)$ . Эффективный алгоритм был разработан Кнудом, Моррисом и Праттом и включает в себя следующие оптимизации:

1. Значение  $\pi[i + 1]$  не более чем на единицу превосходит значение  $\pi[i]$  для любого  $i$ . То есть при переходе к следующей позиции очередной элемент префикс-функции может увеличиться на единицу, не измениться или уменьшиться на какую-либо величину. Поскольку за один шаг значение могло вырасти максимум на единицу, то суммарно для всей строки могло произойти максимум  $(n - 1)$  увеличений и, как следствие, максимум  $(n - 1)$  уменьшений.
2. Необходимо избавиться от явных сравнений подстроки. Для этого постараемся максимально использовать уже вычисленную информацию. Итак, пусть мы вычислили значение префикс-функции  $\pi[i]$  для некоторого  $i$ . Теперь если  $S[i + 1] = S[\pi[i]]$ , то  $\pi[i + 1] = \pi[i] + 1$ . Пусть теперь, наоборот, оказалось,  $S[i + 1] \neq S[\pi[i]]$ . Тогда нам надо попытаться попробовать подстроку меньшей длины, удовлетворяющую свойству. В целях оптимизации хотелось бы сразу перейти к такой (наибольшей) длине  $j < \pi[i]$ , что по-прежнему выполняется префикс-свойство в позиции  $i$ . Действительно, когда мы найдем такую длину  $j$ , то нам

будет снова достаточно сравнить символы  $S[i+1]$  и  $S[j]$  — если они совпадут, то можно утверждать, что  $\pi[i+1] = \pi[j]+1$ . Иначе нам надо будет снова найти меньшее (следующее по величине) значение  $j$ , для которого выполняется префикс-свойство, и так далее. Может случиться, что такие значения  $j$  кончатся — это происходит, когда  $j = 0$ . В этом случае, если  $S[i+1] = S[0]$ , то  $\pi[i+1] = 1$ , иначе  $\pi[i+1] = 0$ <sup>6)</sup>.

Общая схема алгоритма поиска префикс-функции:

- Считать значения префикс-функции  $\pi[i]$  будем по очереди: от  $i=1$  до  $i=n-1$  (значение  $\pi[0]$  равно нулю).
- Для подсчета текущего значения  $\pi[i]$  вводим переменную  $j$ , обозначающую длину текущего рассматриваемого образца. В начале каждого шага положим  $j = \pi[i-1]$ .
- Тестируем образец длины  $j$ , для чего сравниваем символы  $S[i]$  и  $S[j]$ . Если символы отличаются, то уменьшаем длину  $j$ , полагая ее равной  $\pi[j-1]$ , и повторяем этот шаг алгоритма с начала. Если они совпадают, то полагаем  $\pi[i] = j+1$  и переходим к следующему индексу  $i+1$ . Если мы дошли до длины  $j=0$  и так и не нашли совпадения, то процесс перебора образцов прекращается, далее полагаем  $\pi[i] = 0$  и переходим к индексу  $i+1$ .

```
int* PrefixFunction (char* s, int m) {
    int* pi = (int*) malloc (m*sizeof(int));
    for (int i = 0; i < m; ++i) pi[i] = 0;
    for (int i = 1; i < m; ++i) {
        int j = pi[i-1];
        while (j > 0 && s[i] != s[j])
            j = pi[j-1];
        if (s[i] == s[j])
            ++j;
        pi[i] = j;
    }
    return pi;
}
```

Суть алгоритма КМП заключается в том, что по рассчитанной префикс-функции для образца делается вывод о необходимом шаге смещения:

<sup>6)</sup> Головешкин В. А., Ульянов М. В. Указ соч.

Строка	A	B	A	B	A	B	C	A	B	A	B	C	D
Подстрока	A	B	A	B	C	D							
			A	B	A	B	C	D					
									A	B	A	B	C

Рис. 5.4. Поиск подстроки в алгоритме Кнута, Морриса и Пратта (КМП)

В примере дана строка  $T = \text{«ABABABCAVABCD»}$  и образец  $S = \text{«ABABCD»}$ , необходимо найти вход подстроки в строку, используя алгоритм Кнута — Морриса — Пратта. Префикс-функция  $\pi(S)$  равна  $[0, 0, 1, 2, 0, 0]$ .

Строка	A	B	A	B	A	B	C	A	B	A	B	C	D
	A	B	A	B	C	D							

Рис. 5.5. Первый шаг алгоритма КМП

- При первом сравнении совпали 4 элемента. Обращаемся к префикс-функции  $\pi(S) = [0, 0, 1, 2, 0, 0]$  и используем число в четвертой ячейке, равное 2. Это означает, что в просмотренной части строки суффикс длины 2 является префиксом исходной подстроки. Следовательно, можно осуществить сдвиг на  $4 - 2 = 2$  шага.

Строка	A	B	A	B	A	B	C	A	B	A	B	C	D
	A	B	A	B	C	D							
			A	B	A	B	C	D					

Рис. 5.6. Второй шаг алгоритма КМП

- На втором шаге снова совпало пять символов. Обращаемся к префикс-функции  $\pi(S) = [0, 0, 1, 2, 0, 0]$  и используем число в пятой ячейке, равное 0. Это означает, что в просмотренной части строки нет суффикса, который является префиксом исходной подстроки. В этом случае можно осуществить максимальный сдвиг на  $5 - 0 = 5$  шагов.

Строка	A	B	A	B	A	B	C	A	B	A	B	C	D
Подстрока	A	B	A	B	C	D							
			A	B	A	B	C	D					
									A	B	A	B	C

Рис. 5.7. Третий шаг алгоритма КМП

- На третьем шаге все элементы подстроки совпали, соответственно задача нахождения входа подстроки в строку решена.

Точный анализ рассматриваемого алгоритма весьма сложен. Д. Кнут, Д. Моррис и В. Пратт доказывают, что для данного алгоритма требуется порядка  $O(m+n)$  сравнений, что значительно лучше, чем при прямом поиске.

Реализация функции алгоритма на языке C приведена ниже:

```
int KnuthMorrisonPratt (char* t, int n, char* s, int m) {
    int* pi = PrefixFunction (s, m);
    int i = 0;
    int q = 0;
    while (i < n - m + 1) {
        while (q < m && t [i + q] == s [q]) {
            ++q;
        }
        if (q == m) {
            free (pi);
            return i;
        }
        else if (q > 0) {
            i += q;
            q = pi [q];
        }
        else ++i;
    }
    free (pi);
    return -1;
}
```

Существует и прямой алгоритм нахождения подстроки с помощью префикс-функции. Для этого образуем строку  $S + \# + T$ , где символ  $\#$  — это разделитель, который не должен нигде более встречаться. Посчитаем для этой строки префикс-функцию. Равенство  $\pi(i) = m$  означает, что в позиции  $i$  оканчивается искомое вхождение строки  $S$  (только не надо забывать, что все позиции отсчитываются в склеенной строке  $S + \# + T$ ). Таким образом, если в какой-то позиции  $i$  оказалось  $\pi(i) = m$ , то в позиции  $i - m + 1$  строки  $T$  начинается очередное вхождение строки  $S$  в строку  $T$ .

#### 5.1.4. Алгоритм Бойера — Мура

Алгоритм Бойера и Мура считается наиболее быстрым среди алгоритмов, предназначенных для поиска подстроки в строке. Он был разработан Р. Бойером и Д. Муром в 1977 году. Особенность алгоритма в том, что необходимо сделать некоторые предварительные вычисления над подстрокой, чтобы сравнение подстроки с исходной строкой осуществлять не во всех позициях — часть проверок пропускаются как заведомо не дающие результата.

Существует множество вариаций алгоритма Бойера и Мура, рассмотрим простейшую из них, которая основана на следующих принципах:

1. Поиск слева направо, сравнение справа налево. Совмещается начало текста (строки) и шаблона (подстроки), проверка начинается с последнего символа шаблона. Если символы совпадают, производится сравнение предпоследнего символа шаблона и т. д. Если все символы шаблона совпали с наложенными символами строки, значит, подстрока найдена, и поиск окончен. Если же какой-то символ шаблона не совпадает с соответствующим символом строки, применяются два других принципа.
2. Принцип стоп-символа. Если при сравнении символы не совпадают, то шаблон сдвигается так, чтобы совместить несовпадающий символ строки с самым правым таким же символом шаблона. Если стоп-символа в шаблоне нет, то он смещается за этот стоп-символ. Если стоп-символ оказался за другим таким же, используется последний принцип.

Строка	*	*	*	А	*	*	*	*
Подстрока	А	В	А	В				
Следующий шаг		А	В	А	В			

Рис. 5.8. Принцип стоп-символа

3. Принцип совпавшего суффикса. Если при сравнении строки и подстроки совпало один или больше символов, но принцип стоп-символа не работает, то подстрока сдвигается в зависимости от того, какой суффикс совпал.

Строка	*	A	A	B	*	*	*	*
Подстрока	A	B	A	B				
Следующий шаг			A	B	A	B		

Рис. 5.9. Принцип совпавшего суффикса

Пример. Рассмотрим строку T= «ABABABCSABABCD» и подстроку S = «ABABCB»

Строка	A	B	A	B	A	B	C	A	B	A	B	C	B
Подстрока	A	B	A	B	C	B							
			A	B	A	B	C	B					
						A	B	A	B	C	B		
								A	B	A	B	C	B

Рис. 5.10. Поиск подстроки в алгоритме Бойера и Мура (БМ)

Составим таблицу стоп-символов и таблицу суффиксов.

Таблица 5.1

Таблица стоп-символов алгоритма Бойера — Мура

Символ	A	B	C
Сдвиг	3	2	1

Таблица 5.2

Таблица суффиксов алгоритма Бойера — Мура

Суффикс	B	CB	BCB	ABCB	BABCB	ABABCB
Сдвиг	6	6	6	6	6	6

Теперь рассмотрим пример по шагам.

Строка	A	B	A	B	A	B	C	A	B	A	B	C	B
	A	B	A	B	C	B							

Рис. 5.11. Первый шаг алгоритма БМ



- Проверяем справа налево. Вторые два символа не совпали. Смотрим символ *A* по таблице стоп-символов, сдвигаем на  $3-1 = 2$  символа.

Строка	A	B	A	B	A	B	C	A	B	A	B	C	B
	A	B	A	B	C	B							
			A	B	A	B	C	B					

Рис. 5.12. Второй шаг алгоритма БМ

- На втором шаге первые же символы не совпали. Смотрим символ *A* в таблице стоп-символов, сдвигаем на  $3-0 = 3$  символа.

Строка	A	B	A	B	A	B	C	A	B	A	B	C	B
Подстрока	A	B	A	B	C	B							
			A	B	A	B	C	B					
							A	B	A	B	C	B	

Рис. 5.13. Третий шаг алгоритма БМ

- На третьем шаге вторые символы снова не совпали. Смотрим символ *A* в таблице стоп-символов, сдвигаем на  $3-1 = 2$  символа.

Строка	A	B	A	B	A	B	C	A	B	A	B	C	B
Подстрока	A	B	A	B	C	B							
			A	B	A	B	C	B					
							A	B	A	B	C	B	
								A	B	A	B	C	B

Рис. 5.14. Четвертый шаг алгоритма БМ

- На четвертом шаге алгоритма БМ все символы совпали.

Существует упрощенная версия алгоритма Бойера — Мура, алгоритм Бойера — Мура — Хорспула. Отличие состоит в том, что алгоритм БМХ отбрасывает сложную для расчета эвристику совпавшего суффикса и использует модифицированную эвристику стоп-символа. За стоп-символ в этом алгоритме берется символ строки, расположенный непосредственно над последним символом шаблона.

Функция алгоритма Бойера — Мура без использования таблицы суффиксов на языке С имеет вид:

```
int BoyerMoore (char* t, int n, char* s, int m) {
    int i, j, k;
    int s_table [256];
    if (m < n) {
        for (i = 0; i < 256; ++i)
            s_table [i] = m;
        for (i = 0; i < m - 1; ++i)
            s_table [s [i]] = m - i - 1;
        i = m - 1;
        j = m - 1;
        while ((j >= 0) && (i < n)) {
            j = m - 1;
            k = i;
            while ((j >= 0) && (t [k] == s [j])) {
                --k;
                --j;
            }
            i += s_table [t [k]];
        }

        if (k > n - m)
            return -1;
        else
            return k + 1;
    }
    else
        return -1;
}
```

Алгоритм Бойера и Мура оптимален в большинстве случаев, когда нет возможности провести предварительную обработку текста, в котором проводится поиск. Таким образом, данный алгоритм является наиболее эффективным в обычных ситуациях, а его быстроедействие повышается при увеличении подстроки или алфавита. В наихудшем случае трудоемкость рассматриваемого алгоритма  $O(m+n)$ .

Каждый алгоритм поиска позволяет эффективно действовать лишь для своего класса задач, об этом еще говорят различные узконаправленные улучшения. Алгоритм поиска подстроки в строке следует выбирать только после точной постановки задачи.

## *5.2. Поиск в одномерном массиве*

### **5.2.1. Линейный поиск**

Если данные в одномерном массиве не упорядочены, то найти что-либо нас интересующее можно только путем последовательного перебора всех элементов.

В реальных программах «элементами массива» являются, конечно, не только простые значения, но также и более сложные структуры. Та часть элемента данных, которая идентифицирует его и используется для поиска, называется ключом. Остальная часть несет в себе содержательную информацию, которая извлекается и используется из найденного элемента данных.

Поиск значения путем последовательного перебора всех элементов называется линейным поиском. Его трудоемкость, очевидно,  $O(n)$ .

```
int LinearSearch (int* x, int n, int a) {  
    for (int i = 0; i < n; ++i)  
        if (x [i] == a) return i;  
    return -1;  
}
```

### **5.2.2. Бинарный поиск**

Если данные упорядочены, то найти интересующий нас элемент можно значительно быстрее. Алгоритм двоичного (бинарного) поиска основан на делении пополам текущего интервала поиска. В ос-

нове его лежит тот факт, что при однократном сравнении искомого элемента и некоторого элемента массива можно определить, справа или слева следует искать дальше. Проще всего выбирать элемент по середине интервала. Основные идеи алгоритма:

- искомый интервал поиска делится пополам и по значению элемента массива в точке деления определяется, в какой части следует искать значение на следующем шаге цикла;
- для выбранного интервала поиск повторяется;
- когда интервал содержит всего 1 элемент, поиск прекращается;
- в качестве начального интервала выбирается весь массив.

Принцип в чем-то похож на merge sort, на каждом шаге мы также уменьшаем область работы в два раза. Так что очевидно, что алгоритм имеет гарантированную сложность  $O(\log n)$ .

```
int BinarySearch (int* x, int n, int a) {  
    int min = 0, max = n - 1;  
    while (min <= max) {  
        int mid = min + (max - min) / 2;  
        if (a == x [mid]) return mid;  
        else if (a < x [mid]) max = mid - 1;  
        else min = mid + 1;  
    }  
    return -1;  
}
```

Тот же самый принцип используется в структуре «двоичное дерево поиска». Это двоичное дерево со следующими свойствами:

- Оба поддерева — левое и правое — являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла значения ключей данных меньше, нежели значение ключа данных самого узла.
- В то время как у всех узлов правого поддерева того же узла значения ключей данных не меньше, нежели значение ключа данных узла.

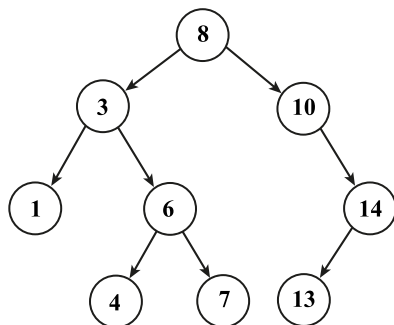


Рис. 5.15. Двоичное дерево поиска

Такое дерево, построенное из элементов одномерного массива, также является одним из способов поиска.

### 5.2.3. Интерполяционный поиск

Поиск происходит подобно двоичному поиску, но вместо деления области поиска на две равные части интерполирующий поиск производит оценку новой области поиска по расстоянию между ключом и текущим значением элемента. Если известно, что  $x$  лежит между  $a_l$  и  $a_r$ , то следующая проверка выполняется примерно на расстоянии  $\frac{x - a_l}{a_r - a_l}(r - l)$  от  $l$ .

Если ключи распределены случайным образом, то за один шаг поиск уменьшает количество проверяемых элементов с  $n$  до  $\sqrt{n}$ . То есть после  $k$ -ого шага количество проверяемых элементов уменьшается до  $n^{\frac{1}{2^k}}$ . Из этого вытекает, что асимптотика алгоритма  $O(\log \log n)$  на равномерных данных. В худшем случае (при экспоненциальном возрастании элементов) время работы алгоритма равно  $O(n)$ . Ниже приведен текст функции интерполяционного поиска.

```

int InterpolSearch (int* x, int n, int a) {
    int l = 0, u = n - 1;
    while (u >= l) {
        int i = l + (u - l) * (a - x[l]) / (x[u] - x[l]);
        if (a < x[i]) u = i - 1;
        else if (a > x[i]) l = i + 1;
        else return i;
    }
}

```

```
    }  
    return -1;  
}
```

Так как разница между  $\log n$  и  $\log \log n$  проявляет себя только на больших  $n$ , на практике оказывается выгодным на ранних стадиях применять интерполяционный поиск, а затем переходить к двоичному поиску

## 6. ЗАДАЧИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ

---

### 6.1. Нахождение минимума в стеке и в очереди за $O(1)$

Задание для лабораторной работы. Модифицировать структуру стека и структуру очереди для нахождения минимума за  $O(1)$ . Решить задачу на нахождение минимумов всех подмассивов длины  $m$  в массиве.

Указания к работе. Описание работы стека приведено в разделе 3.5. Имеющуюся структуру необходимо улучшить так, чтобы можно было найти минимум стека за  $O(1)$ . Для этого необходимо хранить элементы парами. Второе значение в паре будет минимумом в стеке, начиная с данного элемента и ниже. То есть:

$$st \rightarrow A[i].second = \min \{st \rightarrow A[j].first\},$$

где  $j = 0..i$ . При добавлении элемента в стек величина `second` будет равна  $\min \{st \rightarrow A[size].second, a.first\}$ , то есть либо текущему минимуму стека, либо значению нового элемента.

Структура элемента стека может выглядеть так:

```
typedef struct {  
    int first;  
    int second  
} pair;
```

Тогда массив `A` будет массивом таких пар:

```
pair A [max_size];
```

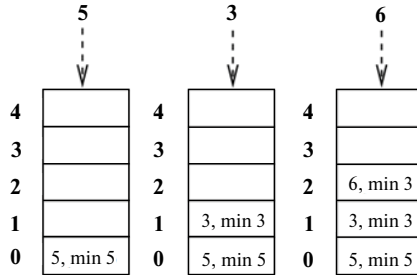


Рис. 6.1. Минимум в стеке за  $O(1)$

Описание работы очереди приведено в разделе 3.6. Для нахождения минимума за  $O(1)$  создадим очередь из двух модифицированных стеков. Добавлять элементы будем в первый стек, а извлекать из второго. При опустошении второго стека (то есть попытке извлечь элемент из пустого стека) перенесем в него весь первый стек. При этом порядок элементов инвертируется, и значения минимумов рассчитаются заново. Общий минимум будет равен  $\min \{st1 \rightarrow A[size].second, st2 \rightarrow A[size].second\}$ <sup>7)</sup>

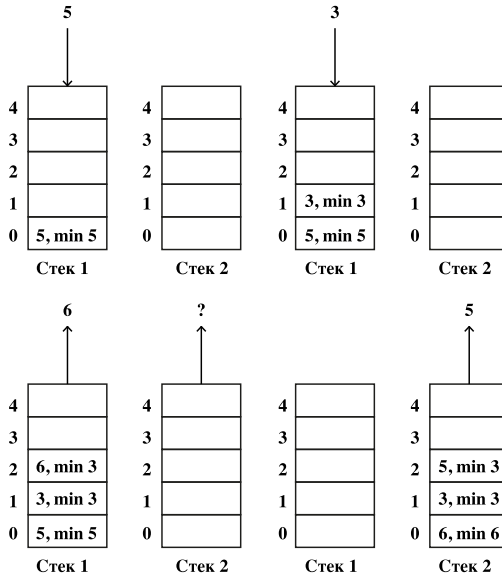


Рис. 6.2. Минимум в очереди за  $O(1)$

<sup>7)</sup> Головешкин В.А., Ульянов М.В. Указ. соч.



Решить с использованием полученной очереди задачу: имеется массив  $[1..n]$ , найти в нем минимумы всех подмассивов длины  $m$ , то есть

$$\min[1..m], \min[2..m+1], \dots, \min[n-m, m].$$

При прямом переборе количество необходимых действий равно  $(n-m) * m$ , то есть на каждом шаге необходимо просматривать целый отрезок длиной  $m$ . Для улучшения асимптотики используем следующий принцип: сначала добавляем первый подмассив длиной  $m$  в очередь, получаем минимум  $[1..m]$ , затем извлекаем из очереди первый элемент и добавляем  $m+1$  элемент, получаем новый минимум — минимум отрезка  $[2..m+1]$ . В итоге понадобится  $(n-m)$  операций для нахождения всех минимумов.

## *6.2. Представление карты дорог с помощью связанных списков*

Задание для лабораторной работы. Написать структуры и функции, реализующие хранение неориентированного графа городов и дорог с помощью связанных списков. Список городов один и начинается с фиктивного головного элемента. Список дорог для каждого города свой. Хранить для каждой дороги пункт назначения. Должны быть реализованы функции добавления города, добавления дороги из одного города в другой. При добавлении дорога дублируется для городов, которые она связывает.

Указания к работе. Описание графов приведено в разделе 3.7, описание связанных списков — в разделе 3.4. При работе с графами можно использовать любой тип связанного списка, поскольку большинство операций в любом случае требуют  $O(n)$  действий.

Рассмотрим пример построения карты дорог с помощью указателей. Эта задача относится к классу задач представления графа.

Пусть  $A, B, C, D$  — названия городов, города соединены между собой дорогами. На рис. 6.3 представлен пример карты дорог между городами:

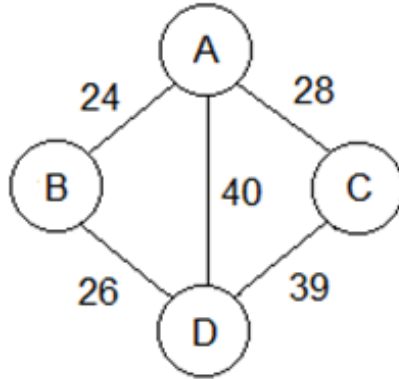


Рис. 6.3. Карта дорог

Чтобы представить этот граф в памяти, можно использовать звенья двух видов: звенья для городов, применяя указатели для связывания города с проходящими через него дорогами, и звенья для дорог для представления ребер графа. Один из способов представления карты можно видеть на рис. 6.4:

**Список городов**

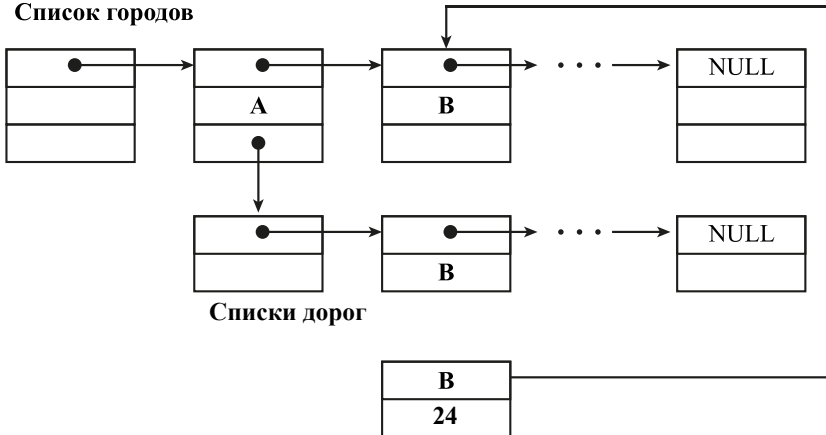


Рис. 6.4. Представление карты дорог с помощью списков

Список городов один и начинается фиктивным элементом. Каждый следующий элемент хранит ссылку на начало списка дорог для города. Каждый такой список также начинается фиктивным элемен-

том. Списки хранят ссылки на непосредственно отдельные структуры, состоящие из указателя на город, куда проложена дорога, и длины дороги.

Пример используемых структур:

```
typedef struct ROAD {
    int length;
    struct CITY *to;
} road_t;
```

```
typedef struct CITY {
    char *name;
    struct list_ROAD *roads;
} city_t;
typedef struct list_ROAD {
    struct ROAD *road;
    struct list_ROAD *next;
} list_road_t;
```

```
typedef struct list_CITY {
    struct CITY *city;
    struct list_CITY *next;
} list_city_t;
```

Список городов можно задать или глобальным указателем на структуру:

```
list_city_t *head_city = NULL;
```

или задать его в функции main:

```
int main () {
    list_city_t *head_city = NULL;
    ...
    return 0;
}
```

Над структурами должны быть реализованы функции добавления города, добавления дороги, функция вывода дорог для выбранного города. Прототипы функций:

```
city_t *newCity (char *name);
road_t *newRoad (int length, CITY *begin, CITY *end);

void printRoads (CITY *city);
```

Внутри функции `newRoad` должна создаваться пара дорог: одна из `begin` в `end` и другая из `end` в `begin`. Новые экземпляры структур внутри функций создаются при помощи команды `malloc` в динамической куче и возвращаются по указателям. Соответственно, при удалении этих структур обязательно использование команды `free`. Пример создания структуры внутри функции:

```
city_t *new_city (char *name) {
    city_t *c;
    if ((c = (city_t*) malloc (sizeof (city_t))) == NULL) {
        perror («malloc»);
        return NULL;
    }
    c->name = name;
    c->roads = NULL;
    return c;
}
```

### *6.3. Максимальная возрастающая последовательность*

Задание для лабораторной работы. Дан массив из  $n$  чисел:  $A[0, 1, \dots, n-1]$ . Требуется найти в этой последовательности строго возрастающую подпоследовательность наибольшей длины. Формально это выглядит следующим образом: требуется найти такую последовательность индексов  $i_1 \dots i_k$ , что:

$$i_1 < i_2 < \dots < i_k, \\ A[i_1] < A[i_2] < \dots < A[i_k].$$

Указания к работе. Пример наибольшей возрастающей подпоследовательности в массиве представлен ниже. Тривиального решения для задачи нахождения такой подпоследовательности нет, поскольку элементы могут быть расположены не друг за другом.

4	2	3	1	5	1	9
---	---	---	---	---	---	---

Рис. 6.5. Наибольшая возрастающая подпоследовательность

Для решения задачи воспользуемся методом динамического программирования. Пусть массив  $D[0..n-1]$  в каждой ячейке  $D[i]$  с номером  $i$  хранит длину наибольшей возрастающей подпоследовательности, оканчивающейся именно в элементе с индексом  $i$ . Массив считается динамически: сначала  $D[0]$ , затем  $D[1]$  и т. д. Итоговым решением задачи будет  $\max\{D[i], i = 0..n-1\}$ .

Пусть текущий индекс  $i$ , и мы хотим посчитать значение  $D[i]$ , а все предыдущие значения  $D[0], \dots, D[i-1]$  уже подсчитаны. Возможны два случая: либо  $D[i] = 1$ , либо  $D[i] > 1$ .

В первом случае текущая подпоследовательность состоит только из числа  $A[i]$ . Во втором случае перед числом  $A[i]$  в подпоследовательности стоит какое-то другое число. Это будет какой-то элемент  $A[j]$  при  $j = 0..i-1$ , такой, что  $A[j] < A[i]$ .  $D[j]$  для этого элемента уже будет подсчитано, соответственно искомое  $D[i]$  будет равно  $D[j] + 1$ . Таким образом,  $D[i]$  можно считать по следующей формуле:

$$D[i] = \max(D[j] + 1) \text{ при } j = 0..i-1 \text{ и } A[j] < A[i]$$

Объединив два случая, получим окончательный алгоритм для вычисления  $D[i]$ :

$$D[i] = \max \left( 1, \max_{\substack{j = 0..i-1 \\ A[j] < A[i]}} (D[j] + 1) \right)$$

Функция поиска наибольшей возрастающей подпоследовательности в массиве принимает в качестве аргументов два массива. Пример такой функции:

```
int arr_max (int* A, int* D, int i) {
    int max = 0, index = 0;
    for (int j = 0; j <= i-1; ++j)
        if (A[j] < A[i] && D[j] > max) {max = D[j]; index = j;}
    return index;
}
```

Возвращаемое значение — индекс  $j$  элемента с наибольшим  $D[j]$ .  
Использовать функция должна так:

$$D[i] = \max(1, D[\text{arg\_max}(A, D, i)] + 1);$$

Для восстановления последовательности можно использовать вспомогательный массив  $P[0..n-1]$  — массив предков. Значение  $P[i]$  будет обозначать тот индекс  $j$ , при котором получилось наибольшее значение  $D[i]$ . Индекс можно получить с помощью вызова функции `arg_max`.

Чтобы вывести ответ, будем идти от элемента с максимальным значением  $D[i]$  по его предкам до тех пор, пока мы не выведем всю подпоследовательность, т. е. пока не дойдем до элемента со значением  $D[j] = 1$ :

```
while (D[i] != 1) {
    printf("%d «, i);
    i = P[i];
}
```

Для проверки работоспособности программы решить вспомогательную задачу: дан массив  $A[0..n-1]$ , требуется раскрасить его элементы в наименьшее число цветов так, чтобы по каждому цвету получалась возрастающая подпоследовательность. Для массива  $\{4, 2, 3, 1, 5, 1, 9\}$  ответом будет 4: подпоследовательности  $\{2, 3, 5, 9\}$ ,  $\{4\}$ ,  $\{1\}$  и  $\{1\}$ . Решение задачи следующее: на каждом шаге для нераскрашенной части массива вызывается поиск наибольшей возрастающей последовательности, затем эта часть окрашивается в очередной цвет.

#### 6.4. *Дерево отрезков*

Задание для лабораторной работы. Написать программу, создающую над массивом дерево отрезков — структуру данных, которая позволяет за асимптотику  $O(\log n)$  реализовать операции нахождения суммы и минимума элементов массива  $A$  в заданном отрезке  $A[l..r]$ .

Указания к работе. Дерево отрезков — это структура данных, которая позволяет эффективно (скорость алгоритма  $O(\log n)$ ) реализовать операции следующего вида:

- нахождение суммы на отрезке  $[l; r]$ ;
- нахождение минимума и максимума на отрезке  $[l; r]$ ;
- изменение элементов (увеличение и уменьшение) на отрезке  $[l; r]$ .

Рассмотрим построение дерева отрезков для нахождения суммы элементов массива  $A[0..n-1]$  на отрезке  $[l; r]$ .

Для построения дерева используется принцип «разделяй и властвуй». Рассчитаем сумму элементов всего массива, т. е. отрезка  $A[0..n-1]$ . Также посчитаем сумму двух частей этого массива:  $A[0..n/2]$  и  $A[n/2+1..n-1]$ . Каждую из этих двух частей также разобьем пополам и посчитаем сумму на их частях и так далее, пока длина текущего отрезка не достигнет 1.

Иными словами, стартуем с отрезка  $[0; n-1]$  и каждый раз делим текущий отрезок на две части, пока он не стал отрезком единичной длины, вызывая затем эту же процедуру от обеих половинок; для каждого такого отрезка храним сумму чисел соответствующего подмассива  $A$ .

Отрезки, на которых мы посчитали сумму, образуют дерево: корень этого дерева — отрезок  $A[0..n-1]$ , а каждая вершина имеет ровно двух сыновей (кроме вершин-листьев, у которых отрезок имеет длину 1).

Дерево отрезков имеет линейный размер и содержит менее  $2n$  вершин: первый уровень дерева отрезков содержит одну вершину, второй уровень — в худшем случае две вершины, на третьем уровне в худшем случае будет четыре вершины и так далее, пока число вершин не достигнет  $n$ . Таким образом, число вершин в худшем случае оценивается суммой  $n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \dots + 1 < 2n$ .

При  $n$ , которое не является степенью двойки, не все уровни дерева отрезков будут полностью заполнены. Например, при  $n = 3$  левый сын корня есть отрезок  $[0..1]$ , имеющий двух потомков, в то время как правый сын корня — отрезок  $[2..2]$ , являющийся листом. Для удобства дополним длину массива до степени двойки. В добавленные элементы массива допишем нули. Итак, дерево отрезков — это двоичное дерево, в каждой вершине которого написано значение заданной функции на некотором отрезке. Высота дерева отрезков равна  $O(\log(n))$ , потому что длина отрезка в корне дерева равна  $n$ , а при переходе на один уровень вниз длина отрезков уменьшается примерно вдвое.

Процесс построения дерева отрезков по заданному массиву можно делать эффективно следующим образом: сначала запишем значения элементов  $A[i]$  в соответствующие листья дерева, затем на их основе посчитаем значения для вершин предыдущего уровня как сумму значений в двух листьях, затем аналогичным образом посчитаем значения для еще одного уровня и т. д. Удобно описывать эту операцию рекурсивно: запускаем процедуру построения от корня дерева отрезков, а сама процедура построения, если ее вызвали не от листа, вызывает себя от каждого из двух сыновей и суммирует вычисленные значения, а если ее вызвали от листа — то просто записывает в себя значение этого элемента массива. Асимптотика построения дерева отрезков составит, таким образом,  $O(n)$ .

Рассмотрим теперь запрос суммы. На вход поступают два числа  $l$  и  $r$ , и за время  $O(\log(n))$  необходимо посчитать сумму чисел на отрезке  $A[l..r]$ . Для этого будем спускаться по построенному дереву отрезков, используя для подсчета ответа рассчитанные ранее суммы в каждой вершине дерева. Изначально мы встаем в корень дерева отрезков. Определим, в какого из двух его сыновей попадает отрезок запроса  $[l..r]$ . Возможны два варианта: отрезок  $[l..r]$  попадает только в одного сына или отрезок пересекается с обоими сыновьями. В первом случае просто перейдем в того сына, в котором лежит отрезок-запрос. Во втором же случае сначала перейдем в левого сына и посчитаем ответ на запрос, а затем — перейдем в правого сына, посчитаем в нем ответ и прибавим к первому ответу. Иными словами, если левый сын представлял отрезок  $[l_1..r_1]$ , а правый — отрезок  $[l_2..r_2]$  ( $l_2 = r_1 + 1$ ), то мы перейдем в левого сына с запросом  $[l..r_1]$ , а в правого — с запросом  $[l_2..r]$ .

По определению фундаментальный отрезок — такой отрезок, для которого существует вершина в дереве, хранящая значение функции на данном отрезке. В нашем случае, функция — это сумма элементов подотрезка массива.

Рассмотрим на примере. Пусть у нас есть дерево отрезков для 8 элементов. Запишем, какой отрезок индексов соответствует каждой вершине:



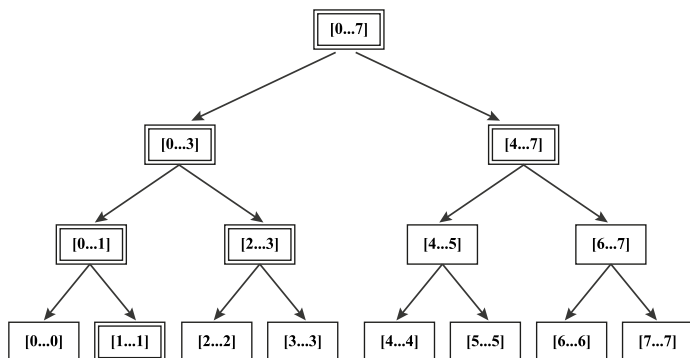


Рис. 6.6. Дерево отрезков для 8 элементов

Рассмотрим, как будет выполняться запрос для отрезка  $[1..5]$ .

Сначала встаем в корень — отрезок принадлежит обоим сыновьям. Значит, нужно разбить его на 2 отрезка:  $[1..3]$  и  $[4..5]$ . Для каждого из них рекурсивно вычислить значение суммы. Отрезок  $[1..3]$  опять принадлежит двум сыновьям. Разбиваем его на 2 отрезка:  $[1..1]$  и  $[2..3]$ . Отрезок  $[1..1]$  принадлежит только правому сыну, спускаемся в него и замечаем, что отрезок  $[1..1]$  — фундаментальный. Берем для него значение суммы из вершины, и поднимаемся до 2 уровня (вершина  $[0..3]$ ). Для левого сына мы уже рекурсивно посчитали необходимую часть суммы, теперь посчитаем для правого. Для этого спускаемся в него. Отрезок  $[2..3]$  — фундаментальный, поэтому берем значение (сумму) из вершины. Возвращаемся в вершину  $[0..3]$  и теперь вычисляем значение суммы для отрезка  $[1..3]$ , так как значение функции уже вычислили для обеих его частей. Возвращаемся в корень и спускаемся в правого сына  $[4..7]$ , подотрезок  $[4..5]$  принадлежит только одному левому сыну, спускаемся в него. Вершине соответствует отрезок  $[4..5]$ , следовательно, он фундаментальный, берем из вершины значение. Возвращаемся в корень и вычисляем ответ.

Этот запрос выполняется за логарифмическое время. Как известно, глубина (количество уровней) дерева из  $n$  вершин равняется  $\log_2 n$ , кроме того, на каждом уровне мы посетим не более 4 вершин, таким образом, мы посетим  $O(\log_2 n)$  вершин.

Итак, обработка запроса суммы представляет собой рекурсивную функцию, которая всякий раз вызывает себя либо от левого сына, либо от правого (не изменяя границы запроса в обоих случаях), либо от обоих сразу (при этом разбивая запрос на два соответствующих подзапроса).

Запрос обновления перестраивает дерево отрезков таким образом, чтобы оно соответствовало новому значению  $a[i] = x$ . Это более простой запрос по сравнению с запросом подсчета суммы. Элемент  $a[i]$  участвует только в относительно небольшом числе вершин дерева отрезков, а именно в  $O(\log(n))$  вершинах — по одной с каждого уровня.

Запрос обновления можно реализовать как рекурсивную функцию: ей передается текущая вершина дерева отрезков, и эта функция выполняет рекурсивный вызов от одного из своих сыновей (от того, который содержит позицию  $i$  в своем отрезке), а после этого — пересчитывает значение суммы в текущей вершине точно таким же образом, как это делалось при построении дерева отрезков.

Хранить дерево отрезков можно так же, как и бинарное дерево: допустим, что корень дерева имеет номер 1, его сыновья — номера 2 и 3, их сыновья — номера с 4 по 7 и так далее. То есть если вершина имеет номер  $i$ , то ее левый сын — это вершина с номером  $2i$ , а правый — с номером  $2i + 1$ .

Такой прием значительно упрощает программирование дерева отрезков: теперь не нужно хранить в памяти структуру дерева отрезков, а можно использовать массив для сумм на каждом отрезке дерева отрезков. Размер этого массива при такой нумерации должен быть  $4n$ .

В результате работы должны быть реализованы три функции: функция построения дерева отрезков, функция нахождения суммы на отрезке и функция обновления сумм в дереве.

### 6.5. Сортировка в графе

Задание для лабораторной работы. Написать программу, которая для данного ориентированного графа перенумеровывает его вершины в топологическом порядке.

Указания к работе. Описание ориентированных графов представлено в разделе 3.7. Топологический порядок в таком графе подразумевает, что каждое ребро выходит из вершины с меньшим номером и приходит в вершину с большим номером. Такой порядок может быть не единственным (например, в пустом графе) или может не существовать вовсе — в случае, если граф содержит циклы.

Для решения задачи используется поиск в глубину на графе. Стратегия поиска в глубину (depth-first search), как следует из ее названия, состоит в том, чтобы идти «в глубь» графа, насколько это возможно. При выполнении поиска в глубину исследуются все ребра, выходящие из вершины,

открытой последней, и поиск покидает вершину, только когда не остается неисследованных ребер — при этом происходит возврат в вершину, из которой была открыта вершина  $s$ . Этот процесс продолжается до тех пор, пока не будут открыты все вершины, достижимые из исходной.

Если при этом остаются неоткрытые вершины, то одна из них выбирается в качестве новой исходной вершины и поиск повторяется уже из нее. Этот процесс повторяется до тех пор, пока не будут открыты все вершины.

В ходе работы алгоритма вершины графа раскрашиваются в разные цвета, свидетельствующие об их состоянии. Каждая вершина изначально белая, затем при открытии (*discover*) в процессе поиска она окрашивается в серый цвет, и по завершении (*finish*), когда ее список смежности полностью сканирован, она становится черной.

Поиск в глубину также проставляет в вершинах метки времени (*timestamp*). Каждая вершина имеет две такие метки: в первой, *in*[ $s$ ], указывается, когда вершина  $s$  открывается (и окрашивается в серый цвет), вторая — *out*[ $s$ ], которая фиксирует момент, когда поиск завершает сканирование списка смежности вершины  $s$  и она становится черной. Эти метки используются многими алгоритмами и полезны при рассмотрении поведения поиска в глубину<sup>8)</sup>.

Реализация функции DFS на языке C:

```
int in [100], out [100], color [100];
int DFS_timer = 0;

void DFS (int** g, int* v, int n, int s) {
    in [s] = DFS_timer++;
    color [s] = 1;
    for (int j = 0; j < v [s]; ++j)
        if (color [g [s] [j]] == 0)
            DFS (g, v, n, g [s] [j]);
    color [s] = 2;
    out [s] = DFS_timer++;
}
```

Массивы *in*, *out*, *color* и переменная *DFS\_timer* заданы глобально для простоты решения. Перед каждым использованием функции они должны быть равны 0. В *in* и *out* сохраняются времена входа и выхода в вершину. Массив *color* хранит цвет вершины. *DFS\_timer* отражает количество выполненных заходов в функцию.

---

<sup>8)</sup> Алгоритмы: построение и анализ. М., 2005

Модифицируем этот алгоритм для вывода топологического порядка следования вершин в графе. По алгоритму к моменту выхода из DFS ( $s$ ) все вершины, достижимые из  $s$ , уже посещены обходом. Следовательно, если мы будем в момент выхода из DFS ( $s$ ) добавлять вершину  $s$  в стек, то в итоге в этом стеке получится топологический порядок графа. На рис. 6.7 приведен пошаговый ход выполнения процедуры DFS с учетом модификации.

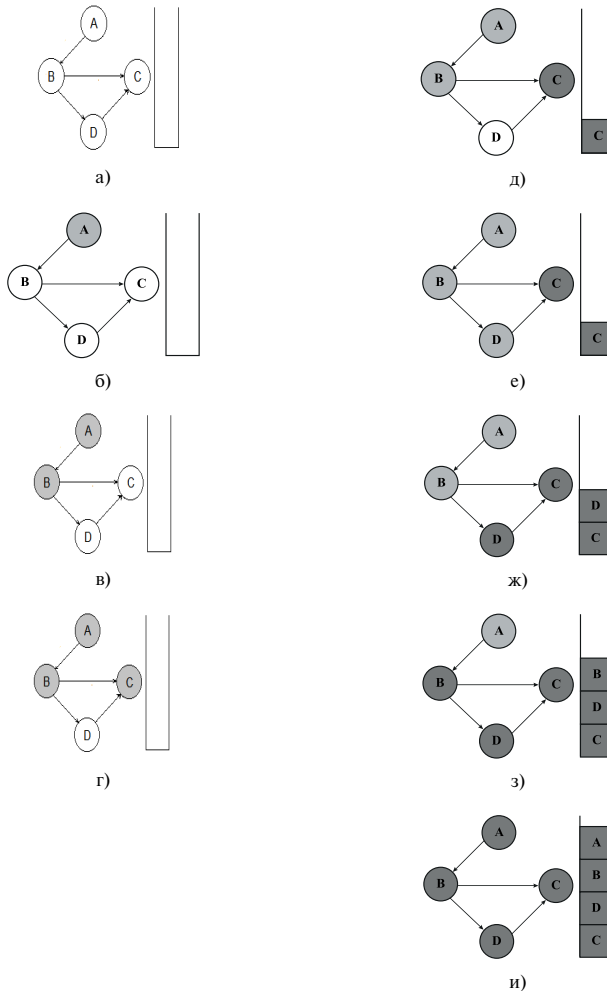


Рис. 6.7. Пошаговый ход выполнения процедуры DFS

- Изначально все вершины белые, а стек пуст. Начнем обход в глубину с вершины *A* (рис. 6.7, *a*).
- Красим вершину *A* в серый цвет (рис. 6.7, *б*).
- Существует ребро из вершины номер *A* в вершину *B*. Переходим к вершине *B* и красим ее в серый цвет (рис. 6.7, *в*).
- Существует ребро из вершины *B* в вершину *C*. Переходим к вершине *C* и красим ее в серый цвет (рис. 6.7, *г*).
- Из вершины *C* нет ребер, идущих не в черные вершины. Возвращаемся к вершине *B*. Красим вершину *C* в черный цвет и кладем ее в стек (рис. 6.7, *д*).
- Существует ребро из вершины *B* в вершину *D*. Переходим к вершине *D* и красим ее в серый цвет (рис. 6.7, *е*).
- Из вершины *D* нет ребер, идущих не в черные вершины. Возвращаемся к вершине *B*. Красим вершину *D* в черный цвет и кладем ее в стек (рис. 6.7, *ж*).
- Из вершины *B* нет ребер, идущих не в черные вершины. Возвращаемся к вершине *A*. Красим вершину *B* в черный цвет и кладем ее в стек (рис. 6.7, *з*).
- Из вершины *A* нет ребер, идущих не в черные вершины. Красим ее в черный цвет и кладем в стек. Обход точек закончен (рис. 6.7, *и*).

Вершины в стек заносятся одновременно с покраской в черный цвет, то есть перед выходом из функции:

```
void DFS (...) {  
    ...  
    color[s] = 2;  
    Push(st, int a);  
    out[s] = DFS_timer++;  
}
```

Стек может передаваться в функцию по указателю или представлять собой глобальную переменную.

Для проверки работоспособности программы решить следующую задачу. Даны  $n$  элементов, для некоторых пар этих элементов известны неравенства  $>$  и  $<$ . Проверить эти неравенства на противоречие, и, если они не противоречивы, вывести элементы в порядке возрастания. При представлении в виде графа задача фактически сводится к проверке этого графа на цикличность. При обходе в глубину сигнала

лом о цикле в графе будет попытка зайти в уже окрашенную в серый цвет вершину. Тогда в тело процедуры нужно добавить такое условие:

```
void DFS (...) {
    ...
    for (int j = 0; j < v[s]; ++j) {
        if (color[g[s][j]] == 0)
            DFS(g, v, n, g[s][j]);
        else if (color[g[s][j]] == 1)
            isCycled = true;
    }
    ...
}
```

Здесь `isCycled` будет глобальной переменной типа `bool`, показывающей, что при запуске DFS граф оказался циклическим.

### 6.6. Поиск подстроки

Задание для лабораторной работы

1. Дана строка  $T$ . Найти наибольшее число  $k$ , и строку  $S$  такие, что  $T$  совпадает со строкой  $S$ , записанной  $k$  раз подряд.

2. Дана непустая строка  $T$ . Найти наибольшее  $k$  такое, что в строке  $T$  есть подстрока  $S$ , являющаяся  $k$ -повторением;  $k$ -повторение — это такая строка  $S$ , что  $S = uuu \dots u$ , где  $u$  — некоторая непустая строка.

Указания к работе

Первая задача решается с помощью префикс-функции. Описание префикс-функции и алгоритм расчета представлены в разделе 5.1.3. Рассмотрим пример к задаче 1:

$S = \text{«abcabcabc»}$ .

Применив префикс-функцию, получим массив  $[0, 0, 0, 1, 2, 3, 1, 2, 3]$ . Если отличное от  $T = S$  решение существует, то последнее значение  $\pi[n]$  префикс-функции не нулевое. И тогда все возможные  $S$  не превышают по длине  $\pi[n]$ . Кроме того, можно отбросить те длины  $i \leq \pi[n]$ , которым не кратна длина  $T$ . Для решения достаточно проверить все подходящие длины  $S$ .

Вторая задача решается с использованием вспомогательного булевого массива. Будем перебирать возможные длины  $L$  повторяющейся подстроки от 1 до  $N$ . На каждой такой итерации построим вспомога-

тельный булев массив  $A$   $[0..N-L]$ , элементы которого определяются как логическое значение выражения  $T[i] = T[i + L]$ . Тогда в этом мас-

сиве идущие подряд  $j$  единиц будут соответствовать  $1 + \left\lceil \frac{j}{L} \right\rceil$  повторениям подстроки длины  $L$ . Ответом на задачу соответственно будет максимальное такое повторение по всем  $L$ .

Для примера возьмем ту же строку  $S = \text{«abcabcbabc»}$ . Тогда  $L = 1..3$ . Вспомогательный массив  $A$  будет равен  $\{0, 0, 0, 0, 0, 0, 0, 0\}$ ,  $\{0, 0, 0, 0, 0, 0, 0\}$  и  $\{1, 1, 1, 1, 1, 1\}$ . Ответ, очевидно,  $1 + \left\lceil \frac{6}{3} \right\rceil = 3$ .

## *Библиографический список*

---

1. Вирт Н. Алгоритмы и структуры данных : пер. с англ. / Н. Вирт. М. : Мир, 1989.
2. Макконнел Дж. Основы современных алгоритмов : пер. с англ. / Дж. Макконнел. М. : Техносфера, 2006.
3. Седжвик Р. Фундаментальные алгоритмы на С. В 5 ч. Ч. 5. Алгоритмы на графах. М. : ДиаСофт, 2003.
4. Алгоритмы и олимпиадное программирование // MAXimal: [сайт]. Режим доступа: <http://www.e-max.ru/algo>.



## *ОГЛАВЛЕНИЕ*

1. АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ.....	3
1.1. Классификация структур данных.....	3
1.2. Операции над структурами данных.....	5
1.3. Понятие алгоритма обработки данных .....	6
1.4. Представление алгоритмов .....	7
2. АНАЛИЗ СЛОЖНОСТИ АЛГОРИТМОВ .....	12
2.1. Классы сложности алгоритмов.....	12
2.2. Методы оценки сложности алгоритмов .....	15
3. СТРУКТУРЫ ДАННЫХ.....	19
3.1. Типы данных .....	19
3.2. Массив .....	21
3.3. Строка.....	24
3.4. Связный список .....	24
3.5. Стек.....	29
3.6. Очередь .....	36
3.7. Граф .....	39
3.8. Дерево .....	44
4. АЛГОРИТМЫ СОРТИРОВКИ.....	48
4.1. Сортировка выбором .....	50
4.2. Обменная сортировка .....	51
4.3. Сортировка вставкой .....	53
4.4. Сортировка слиянием .....	55
4.5. Быстрая сортировка .....	58
4.6. Пирамидальная сортировка.....	60
4.7. Внешняя сортировка.....	64
5. АЛГОРИТМЫ ПОИСКА.....	68
5.1. Поиск подстроки в строке .....	68
5.2. Поиск в одномерном массиве.....	82

---

6. ЗАДАНИЯ ДЛЯ ЛАБОРАТОРНЫХ РАБОТ .....	86
6.1. Нахождение минимума в стеке и в очереди за $O(1)$ .....	86
6.2. Представление карты дорог с помощью связанных списков .....	88
6.3. Максимальная возрастающая последовательность .....	91
6.4. Дерево отрезков .....	93
6.5. Сортировка в графе .....	97
6.6. Поиск подстроки .....	101
БИБЛИОГРАФИЧЕСКИЙ СПИСОК .....	103

*Учебное издание*

**Селиванова** Ирина Анатольевна,  
**Блинов** Владислав Андреевич

**ПОСТРОЕНИЕ И АНАЛИЗ  
АЛГОРИТМОВ ОБРАБОТКИ ДАННЫХ**

Редактор *Т. Е. Мерц*  
Компьютерный набор *В. А. Блинова, И. А. Селивановой*  
Компьютерная верстка *Е. В. Суховой*

Подписано в печать 22.06.2015. Формат 60×84 1/16.  
Бумага писчая. Плоская печать. Усл. печ. л. 6,28.  
Уч.-изд. л. 5,3. Тираж 50 экз. Заказ 166.

Издательство Уральского университета  
Редакционно-издательский отдел ИПЦ УрФУ  
620049, Екатеринбург, ул. С. Ковалевской, 5  
Тел.: 8 (343) 375–48–25, 375–46–85, 374–19–41  
E-mail: rio@urfu.ru

Отпечатано в Издательско-полиграфическом центре УрФУ  
620075, Екатеринбург, ул. Тургенева, 4  
Тел.: 8 (343) 350–56–64, 350–90–13  
Факс: 8 (343) 358–93–06  
E-mail: press-urfu@mail.ru

*Для заметок*

